

T H E B E S T O F



MacTutor

The Macintosh Programming Journal

InitCursor;
case mouseDown:
domouse(&theEvent);
break;

Begin
SystemTask;
DoIt: = GetNextEvent(
End;

CLR.W -(SP)
MOVE.L
PEA
FindWindow
Point(A5), -(SP)
WWindow(A5)

CLR.W -(SP)
MOVE.L
PEA
FindWindow
Point(A5), -(SP)
WWindow(A5)

Begin
System Task;
DoIt: = GetNextEvent(
End;

InitCursor;
case mouseDown:
domouse(&theEvent);
break;

CLR.W -(SP)
MOVE.L
PEA
FindWindow
Point(A5), -(SP)
WWindow(A5)

VOLUME I

The Best of MacTutor

The Macintosh Programming Journal

Vol. 1

Edited by
David E. Smith



Published by
MacTutor™

© 1986 by MacTutor, Anaheim, CA. 92807

This book was created on a Macintosh Plus Computer
using a LaserWriter Plus and Aldus Pagemaker Software.

Cover design by Shelton Advertising, Orange, CA 92667

All rights reserved. No part of this book
may be reproduced, in any form or by any means,
without permission in writing from the publisher.

Printed in the United States of America
by California Offset Printers, Glendale, CA 92104

10 9 8 7 6 5 4 3 2 1

ISBN 0-9617544-0-0



MacTutor

The Macintosh Programming Journal

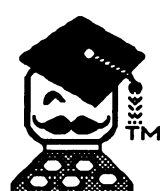
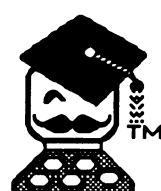
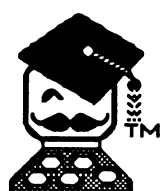
P.O. Box 400
Placentia, CA 92670
(714) 630-3730

1240 Van Buren, #105
Anaheim, CA 92807

A special thanks to Laura Smith and Laurel Galvan-MacLean for their help in preparing the final manuscript from all the many MacTutor articles from our first year of publishing.

A grateful thank you to the MacTutor contributing editors and authors whose articles appear in this book. Without their willingness to share the Macintosh technology, there would be no MacTutor.

And finally, a warm and sincere vote of thanks to the Editorial Board of MacTutor, present members Bob Denny, Dave Kelly, Jörg Langowski, and past board member Chris Derossi, all of whom have contributed to the lion share of this book and helped MacTutor get off the ground with their unselfish research into Macintosh programming.



Contents

Best of MacTutor, Vol. 1

December 1984 Through
December 1985

Contents	v-vi	Pascal Language	226-297
Introduction	vii-viii	<i>MacPascal Arrives!</i>	226
Assembly Language	1-120	<i>Introduction to QuickDraw</i>	228
<i>A Simple Window Program</i>	1	<i>QuickDraw Does Regions</i>	231
<i>Icons as Resources</i>	6	<i>Ports O' Call in QuickDraw</i>	237
<i>Double-Clickable Start-up Ad</i>	12	<i>InLine Traps Create Windows</i>	241
<i>Modeless Dialogs and DA Support</i>	20	<i>Advanced Mac'ing - Pascal Generic Proc</i>	246
<i>Animation Example</i>	31	<i>InLine Traps Simulate OS Calls</i>	248
<i>Jasik Hypes MacNosy</i>	37	<i>Advanced Mac'ing - I/O Completion</i>	250
<i>A MicroFinder Utility</i>	40	<i>Reading Paint Files from Pascal</i>	253
<i>Resource Formats in Assembly</i>	45	<i>Custom Dialog Box for Input</i>	256
<i>Communications Primer</i>	48	<i>All About Printing</i>	259
<i>PrLink Source for MacAsm</i>	53	<i>Dial-a-Fortune Apple Talk Client</i>	265
<i>Ask Prof. Mac - Text Edit</i>	61	<i>Pascal Icons Revealed</i>	272
<i>Electrical Mac - Mac Memory Explained</i>	65	<i>The Amazing Pic to Clip Utility</i>	273
<i>Debugging Techniques in Tmon</i>	68	<i>A Resource Utility DA in TML</i>	278
<i>Ask Prof. Mac - Screen Sleep</i>	71	<i>Christmas Graphics</i>	290
<i>Grow Window in MacAsm</i>	77	Basic Language	298-360
<i>Sound Lab - Midi Connection</i>	83	<i>Alphabet Soup</i>	298
<i>Electrical Mac - Mac PALs & Serial Ports</i>	85	<i>Anybody Know What Date it is?</i>	301
<i>Ask Prof. Mac - Tech Questions</i>	92	<i>May I have Your Order Please?</i>	304
<i>Talking Mac</i>	95	<i>What Light Through Yonder Window</i>	307
<i>Sound Lab - Midi Connection Part II</i>	102	<i>Rectangle Utility for Controls</i>	310
<i>Ask Prof. Mac - HFS File Structure</i>	107	<i>Poke Screen Memory Directly</i>	313
<i>Advanced Mac'ing - DissBits Subroutine</i>	111	<i>The Art of Clipboarding</i>	315
C Language	121-225	<i>Standard File Dialog from Basic</i>	317
<i>A New World</i>	121	<i>Printing Techniques</i>	320
<i>Application Template in C</i>	124	<i>Special Projects - MS Basic Structure</i>	323
<i>Planning For Desk Accessories</i>	130	<i>Fat Bits Approach to Cursor Editing</i>	328
<i>The Mac Window Technology</i>	133	<i>Reading Paint Files</i>	332
<i>Window Dynamics</i>	138	<i>Special Projects - Rescue Protected Basic</i>	336
<i>Text Edit & Scrolling Dynamics</i>	142	<i>Printing Techniques for Basic</i>	339
<i>Custom File Type Dialog Box</i>	151	<i>Library Routine to Eject the Disk</i>	343
<i>MacPaint File Formats in C</i>	158	<i>Play Icon Concentration</i>	347
<i>Function Resources</i>	162	<i>Special Projects - MacPaint Simulator</i>	351
<i>Using the Vertical Retrace Manager</i>	168	Forth Language	361-406
<i>Toolbox Tips - Arrows From C</i>	174	<i>Forth Goes SANE</i>	361
<i>Dial-a-Fortune Apple Talk Server</i>	177	<i>A Forth Decompiler</i>	364
<i>C Glue Routines for Filter Procs</i>	190	<i>Traps and so FORTH</i>	368
<i>Programmer's Guide to NetWorking</i>	194	<i>Controls from Forth</i>	370
<i>Desk Accessories in C</i>	204	<i>Disk Directories & File Editing</i>	373
<i>PICT Rotation with CopyBits</i>	210	<i>Forth Questions & Answers</i>	377
<i>Try Pop-Up Menus!</i>	218	<i>Principals of Text Editing</i>	379
		<i>Recovering Lost Files</i>	383
		<i>InLine Code Speeds MacForth</i>	387

<i>Forth Blocks to Ascii Text</i>	391		
<i>Solving Systems of Linear Equations</i>	395	<i>Modula 2 Language</i>	456-479
<i>Curve Fitting, Part II</i>	399		
<i>Curve Fitting, Part III</i>	403	<i>Modula's History</i>	456
Fortran Language	407-425	<i>Matrix Multiplication</i>	458
		<i>Towers of Hanoi, Part I</i>	461
<i>Random Generator in MacFortran</i>	407	<i>Towers of Hanoi, Part II</i>	465
<i>Fractals in Fortran 2.1</i>	414	<i>Anchored Variables</i>	468
<i>Shell Application</i>	419	<i>Using the Alternate Screen Buffer</i>	471
		<i>Debugging with MacsBug</i>	475
Lisp Language	426-455	APL Language	480-483
<i>An Object Oriented Language</i>	426	<i>A Beginner's Look at Mac APL</i>	480
<i>Introduction to ExperLisp</i>	428		
<i>Functions in Lisp</i>	431	<i>The Mousehole Report</i>	484-502
<i>Cons Cells and QuickDraw</i>	436		
<i>Iteration Techniques</i>	440	<i>Author's Index by Richard Schroedel</i>	503-504
<i>Recursion and Windows</i>	444		
<i>Windows and Tic-Tac-Toe</i>	447		
<i>3-D Rotations</i>	452		

Note: Two articles were unintentionally left out and will be included in Volume 2:
 Asm: 1-9 "Wired For Sound" by Chris Yerga
 Pascal: 1-13 "Mondrian Shows Dissolve Effects" by Mike Morton



Introduction

The Making of MacTutor



David E. Smith
Editor & Publisher

Computer Beginnings

My first real interest in computers didn't come along until computers got small and personal. For me, that happened in January 1975 when the MITS Altair 8800 appeared on the cover of Popular Electronics. At that time I was teaching High School Mathematics in San Diego with Dr. Melvin Zeddies, who immediately ordered several of the kits for the Independent Studies Department of Moorsee High school. When I later built my own computer that year for my classroom in Blythe California, I entered the new world of personal computing. At first the price was \$439, but by the time I ordered it, at the end of the summer special, it had risen to \$995, for an 8K machine.

The Age of Altair

Things moved quickly from that first summer of the Altair. Byte magazine had started publishing in September. Dick Heiser opened the first retail computer store in Santa Monica. MITS sold 10,000 Altair computers before competition began to make inroads with the IMSAI 8080, Digital Group, Polymorphic and a new "typewriter" computer by Sol Libes of Processor Technology.

In 1976, the Altair bus type computers, named S-100 by the jealous competition, began its glory years. The First World Altair Conference introduced the idea of a computer specific consumer show and we all met David Bunnell, Ted Nelson and Bill Gates for the first time. David was editing the newsletter for MITS and Bill was showing off his 8K MITS Basic interpreter for the Altair. Ted Nelson thrilled us all with predictions of personal computers in every home and we snapped up his crazy Computer Lib book. David and Bill certainly have come a long way since then. David Bunnell now heads up Mac World among other things, while Bill Gates took his 8K Basic and created a software empire at Microsoft. Me, I was trying to get my Pertec disk drive to stop crashing long enough to teach some tenth graders how to plot algebraic equations!

The Age of Home Computing

I quickly learned that the S-100 type systems were not quite ready for the rigors of the classroom, and neither was I. So I moved to LA to get into Software Engineering for Lanny Lewyn at a company that made Heart Pacemakers. Meanwhile, the Apple II, Commodore PET and the Radio Shack TRS-80 began to spell the end of the S-100 era. This was the period of the great "Red Book" that provided the only real documentation for the new funny looking Apple II. Introduced at the First West Coast Computer Faire in 1977, it seemed

like a toy. I distained it and bought a PET, which seemed like a much better machine at the time. During 1978, Tandy became the market leader selling 100,00 computers based on the Z-80, while Commodore sold 25,000 PET computers and little garage shop Apple sold 20,000 Apple II's.

Somewhere along the way, Apple Computer figured out the magic formula for turning a garage operation into a real company. The talk at the shows was that Steve Jobs had done the unthinkable and sold all but 10% of his company to outside investors. We all lamented how sad it was that poor Steve had given up so much! Little did we know that all the others who retained ownership would quickly disappear. Gone were the Xitan, PUP-1, Equinox 100, Sol, Altair, Poly-88, Noval and one that never even got built, the infamous ECD 6502 computer, advertised as the "Cat's Meow".

The Age of Apple

Another smart move the young Apple made was to establish a dealer base. While Commodore missed deadlines and generally frustrated dealers, Apple quietly built-up a nationwide chain of computer stores that made it the most readily available computer on the market by the end of 1978. I saw the writing on the wall and sold my PET just before the price dropped and bought an Apple II.

The year 1979 began the golden age of the Apple II, as home computing made a brief appearance. TI tried to give away razor computers while selling the software blades. Look-alike machines sprang up from Atari. Hobby groups around the country rose up to support the Apple machine. The most famous of these was Call A.P.P.L.E., edited by Val Golding. This was the indispensable source of technical information on the Apple. At this time, Byte and Creative Computing and other computer magazines were trying to cover all the different computers in each issue. I wanted a magazine devoted to my computer, the Apple II. So I invented one. And The Apple Shoppe was born. This was the first computer specific magazine to be published that was not sponsored by a computer club. The Apple Shoppe was written almost entirely by myself and my wife Laura and I ran the business out of our home, building it up to 4,000 subscribers before the strain of it all took its toll. The Apple Shoppe covered a single computer with each column a different programming topic. This format soon had a host of copycats when Nibble Magazine started up using the identical format. During the next two years, Apple became the darling of the computer world with Visi-Calc and UCSD Pascal and everyone watched and waited as TI blundered and IBM planned.

Good-by to Innovation!

When IBM came into the scene, the whole business changed. Computer stores fired the technical types (I was one!) and hired MBA students who promised to sell, sell, sell, not to those computer hobbyists who hung around the stores, but to the new "get rich quick" market of "The Business Computer". The Apple Shoppe went away, IBM came out with the boring PC, and personal computers began to lose interest for me and many others as IBM made the whole thing a commodity for businessmen. I got out of personal computers and went to work for Hughes Aircraft. Dick Heiser sold his computer store. Everyone predicted the death of Apple Computer, Val Golding stopped editing Call A.P.P.L.E., and Ted Nelson stopped talking about the future. It was the dark ages of personal computing.

SmallTalk to the Rescue!

The light again began to shine when rumors appeared that Apple would bring out a SmallTalk machine. Many of us had seen and heard Alan Kay promote his DynaBook concept and while CP/M, UCSD P-System, and DOS debated the merits of operating systems, what we all really wanted was something like SmallTalk. Well, Apple came out with it and we all rushed to the store only to find out that it had a \$10,000 price tag! So while a few hearty souls shelled out the ten grand, the rest of us drooled and waited. Then the rumors began of a secret development group within Apple that would build an affordable Lisa computer, and while Lisa flopped, the pressure increased for the unknown machine to make its appearance.

An Appliance Computer

When the Macintosh was introduced, it quickly was promoted as some kind of computing appliance for the uneducated masses that were thought could not understand what a computer was for. This ridiculous notion had been tried previously by TI with the notorious 99/4 computer and I thought their experience would have been sufficient example. The Macintosh was the first personal computer not to have a

programming language built into it and Apple offered no way to program it except by buying the defunct Lisa system, which Apple began to discount heavily to developers. As a result of this, numerous magazines started up which continued to promote the notion of a non-technical computer. This began the age of "fluff" magazines as each tried to outdo the other in color, graphics and bland editorial reviews.

No Fluff!

The straw that broke the camel's back for me was when MacWorld published a feature article on how to clean your mouse! That was simply too much. I could stand it no longer. I rejected Apple's unbelievable position that you had to buy two computers to program one, and determined to publish an old fashioned technical Journal that promoted programming ON the Mac, FOR the Mac! And thus MacTutor was born. A post on the mousehole BBS asked if anyone would be interested in writing for the new magazine and several prolific mousehole members responded. Bob Denny agreed to write on C, Dave Kelly on Basic, and Chris Derossi would take Pascal. Jörg Langowski had left several Forth posts from another board, and he agreed to continue writing on Forth. Later I found out my future board member lived in Germany! So our Editorial Board took on an international flavor. At the November 1984 Comdex, the first issue of MacTutor was displayed at the Apple Booth. Dan Cochran of Apple Computer allowed us to pass out sample copies at the last minute and soon 200 subscribers began receiving The Macintosh Programming Journal. Jim Fitzsimmons of Mac America took over our distribution and soon we were widely available in computer stores and book stores. As they say, the rest is history and this Best of Book combines all the issues from our first year of publication. As we look forward to our third year of continuous publication, the Mac programming world just gets more and more interesting. Soon we will all be running 68020 based Macs with upwards of 4 megabytes and the power of a Vax. Looks like we are entering the golden age of the Macintosh. And MacTutor will continue to be there, promoting the Macintosh Programming Technology.

David E. Smith

October 1, 1986



*All the programs published in this volume
are available on disk. Contact MacTutor
about ordering a set of the "Best of
MacTutor" source code disks.*

Assembly Language Lab

A Simple Window Program



David E. Smith
Editor & Publisher
MacTutor Vol. 1 No. 1

Introducing the Mac Assembler

Welcome to the Assembly Language Lab. In this column, we will be discussing programming techniques and applications using the Macintosh Assembler, due to be released. With the assembler, Apple has at last provided a complete stand-alone development system for the Macintosh computer.

The Mac assembler was written by Bill Duvall under contract from Apple and is an excellent product. It includes an editor, assembler, linker and debugger in addition to a number of useful utilities for manipulating icons, fonts and resources. All of the Macintosh toolbox and operating system trap calls are fully supported with macros and equate files. This month's column is based on the August 29, 1984 pre-release version of the assembler/editor system.

Editor

The editor is nicely integrated with the whole system and provides a very flexible tool for examining and modifying any text file, including BASIC, MacWrite and C as well as assembly source files. The editor is fully disk-based, fast, and outputs text files that can be read directly by MacWrite. Of course, it does not provide formatting capability for word processing as MacWrite does, but it is very efficient for source code editing, or for examining any text file; in this regard, it is the closest thing we have to a universal editor on the Macintosh.

Editor Files

A number of text files are created by the editor, as shown in Fig. 1. These are:

.asm	Assembler source file input
.files	List of source file names
.link	Linker input instructions
.job	Exec type input file
.R	Resource maker input file

Each of these file extensions indicates a different type of text file created by the editor for use with the other development system tools. The '.files' type is simply a file containing a list of assembly source code file names. This allows a batch of files to be assembled at one time. The '.job' type is similar to the old EXEC file on the Apple II in that it is a list of assembly and link commands that are executed one

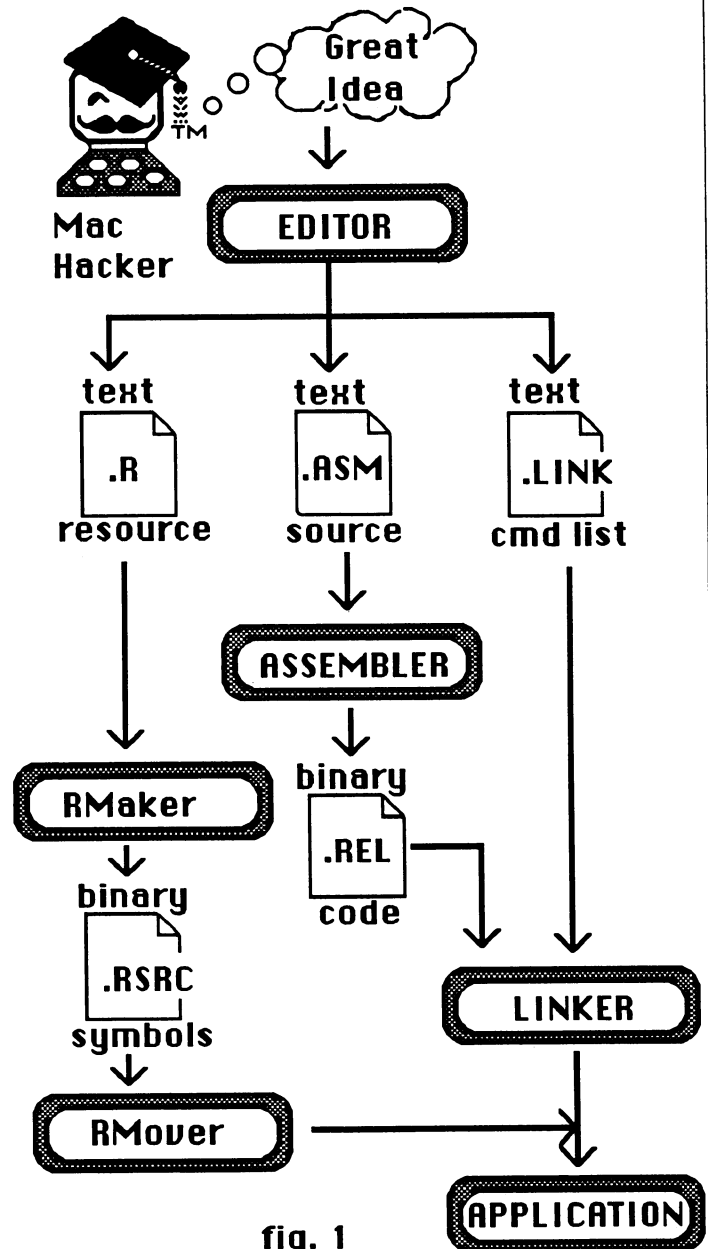


fig. 1

after another as if they were selected with the mouse. The '.R' type file is unique to the Macintosh. This is a text file of resource descriptions that are compiled into a binary format that can be inserted into the system resource file or an application resource. The '.link' type provides a list of linker instructions to tell the linker how to create a stand-alone application from the relocatable output of the assembler.

Assembler

The assembler takes text files with the '.asm' extension and produces relocatable code files with the '.rel' extension. These files are not runnable, however. To be runnable, they must first be linked together with the linker. Other files created by the assembler include the listing file and an error file. One non-standard feature is the directive '.dump' that creates a symbol table file from equate files. The purpose of this is to allow a method of compacting the huge library of system equates that Apple has created as part of the Macintosh development effort on the Lisa. These files take up a considerable amount of room. But with the '.dump' directive, and a utility to pack the symbols file called PackSym, this overhead is greatly reduced.

Linker

The linker is the heart of the development system. It is what actually puts together an application program for you. Macintosh files are composed of two sections called a "Resource Fork" and a "Data Fork". The resource part of an application file includes icon and font definitions, window-making instructions and the actual binary code of the program itself. The data part of the file is defined by the application program and normally is empty when the program starts up. The linker knows about resource files and how to pack the code into the application resource fork.

A number of important memory considerations involve the linker. Macintosh has a memory manager and segment loader that control where in memory different parts of an application program should be placed. Data structures defined in the source code by use of the "DC" directive are stored on the application heap along with the program code in segments. Variable storage defined by the "DS" directive are relocated by the linker and stored in an applications globals area which is referenced to register A5, as shown in Fig. 2. The start of this applications globals area is set in the linker input instructions with the GLOBALS command.

The linker input file defines the segments in which the program code will be loaded by the linker. After the program is running, the segment manager can then dispose of unused segments. The manner in which the linker input file is put together determines which code files are associated with which segment. The linker also allows precompiled resource files and predefined data files to be linked to the application code as well. This allows all the files, both resource and data, that are required by a program to be linked together into the resource fork and data fork of the application. In this manner, the user sees only a single file on the disk; a file that in actuality is composed of several files. The system file is an example of a single file composed of many parts.

Getting Started

In this month's column, we get our feet wet with a program to create a window on the Macintosh. Since our program will call a number of toolbox routines, a few words about macros

MAC MEMORY MAP (128K)

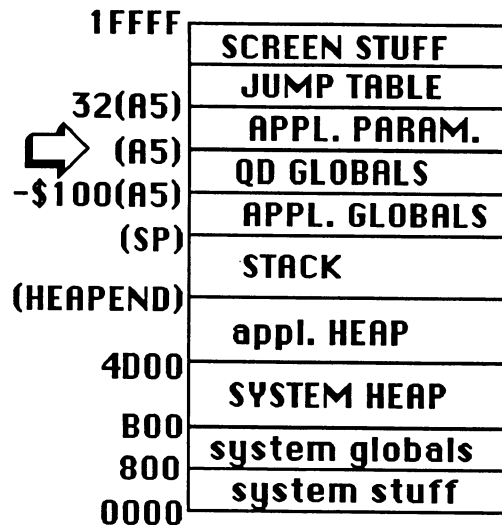


fig. 2

are in order. The Macintosh assembler uses both a Lisa type of macro and a style unique to the Mac. The Mac style is designated by the word MACRO followed by the macro name, and a vertical bar, '|', that signifies the end of the macro. Be careful not to confuse this with a number one.

Our first step is to define the trap macros for the toolbox and operating system routines. This is done by setting the toolbox routine name equal to a trap word constant using the "DC.W" directive. All instructions found in memory that begin with \$A are unimplemented instructions that cause the 68000 to "trap" to a special routine to handle the exception. Apple has taken advantage of this arrangement to extend the 68000 instruction set by a whole series of toolbox routines that are called as a result of this "1010" trap.

The Mac Does Windows

```
; EXAMPLE ASSEMBLY PROGRAM
; WINDOWS2.5 (MacTech 1-1)
; VERSION 13 OCT 84
; (C) 1984 MacTec by David E. Smith
```

```
; Macro subset for Toolbox stuff
```

```
MACRO _InitGraf = DC.W $A86E|
MACRO _InitWind = DC.W $A912|
MACRO _NewWindow = DC.W $A913|
MACRO _setport = DC.W $A873|
MACRO _InitFont = DC.W $A8FE|
MACRO _InitMenu = DC.W $A930|
MACRO _InitDialog = DC.W $A97B|
MACRO _TEInit = DC.W $A9CC|
MACRO _Initpack = DC.W $A9E5|
MACRO _FlushEvents = DC.W $A032|
MACRO _InitCursor = DC.W $A850|
```

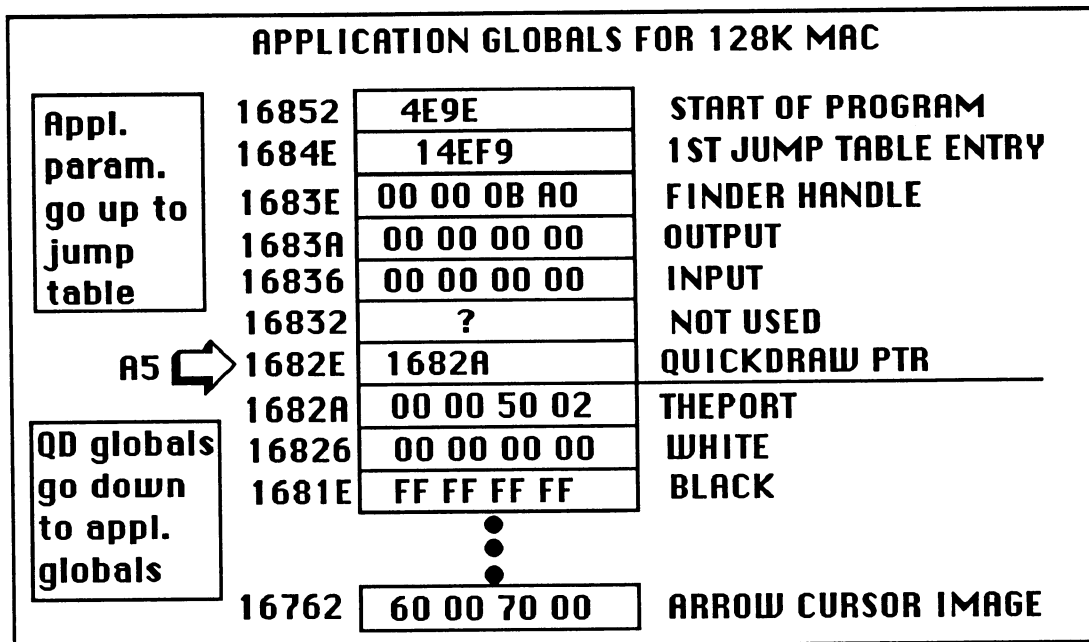


fig. 3

```

MACRO _GetNextEvent =      DC.W $A970|
MACRO _FrameRect =        DC.W $A8A1|

; DECLARE LABELS EXTERNAL

XDEF  START                ; required for linker

; LOCAL EQUATES

MouseDown      equ         1
AllEvents equ    $0000FFFF

; MAIN PROGRAM SEGMENT

DC.B    'MINE'              ;find start of program

; --- SAVE THE WORLD -----

START:  MOVEM.L  D0-D7/A0-A6, -(SP)
        LEA     SAVEREGS,A0
        MOVE.L  A6,(A0)      ; local var
        MOVE.L  A7,4(A0)     ; stack ptr

Since a window is a grafport, our first task is to initialize
quickdraw. To do this, we must supply a memory location
where the quickdraw globals can be stored. In Fig. 3 we see
how the quickdraw globals are stored relative to A5, between
the application parameters, and the application globals.

; ---INITIALIZE ALL MANAGERS-----

; SET UP QUICKDRAW GLOBALS

PEA -4(A5)      ;push qd global ptr
_InitGraf ;init quickdraw global

```

The quickdraw globals point to the current drawing port, as shown in Fig. 4. From A5, the application globals area is found. Then the pointer to the quickdraw globals, followed by the quickdraw globals themselves. Finally, the first entry in the quickdraw globals is a pointer to the current port defined by the current grafport data structure. The quickdraw globals are defined in Fig. 5. The grafport is described in "Inside Macintosh" and is too long to be included here.

Our next task is to define the remaining toolbox managers:

POINTERS AND HANDLES

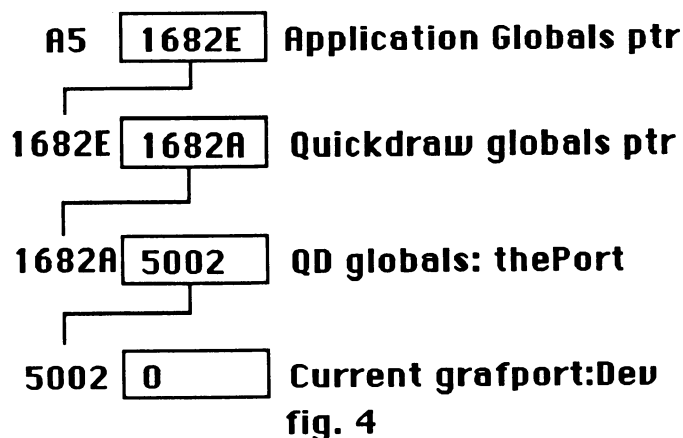


fig. 4

QUICKDRAW GLOBALS

<u>symbol</u>	<u>offset</u>	<u>item type</u>
theport	-0	; ptr to port
white	-8	; pattern
black	-16	; pattern
gray	-24	; pattern
ltgray	-32	; pattern
dkgray	-40	; pattern
arrow	-108	; cur bit map
screenbits	-122	; scr bit map
rndseed	-126	; random #

Fig. 5

---- SET UP REMAINING MANAGERS ---

```
_InitFont      ; init font manager
_InitWind      ; init window manager
_InitMenu      ; init menu manager
```

```
CLR.L  -(SP)    ; kill the restart
_InitDialog      ; init dialog manager
```

```
_TEInit      ; init text edit (ROM)
```

```
MOVE.W #2,-(SP) ; set-up
_Initpack      ; init package mgr
```

There are two types of windows; those defined on the heap, as we are doing here, and those defined as resources, which we will cover next month. The window definition is defined on the heap by the _NewWindow routine, and is not relocatable. Hence, the heap can not be managed as effectively, but our program is small enough for this not to be a problem. Like all toolbox routines, we must imitate the Lisa Pascal calling sequence by pushing the necessary variables on the stack before actually calling the toolbox routine. The first item pushed is always space for any returned value, in this case, the pointer to the new window.

-- SET UP NEW WINDOW ON HEAP ---

```
CLR.L -(SP)      ;return window ptr
CLR.L -(SP)      ;window record ptr.
PEA WBOUNDS      ;window rectangle
PEA WINDTITLE     ; window title
MOVE.W #$100,-(SP) ; true = visible
MOVE.W #0,-(SP)   ; doc type window
MOVE.L #-1,-(SP)  ; window in front
```

```
MOVE.W #$100,-(SP) ; true=closebox
MOVE.L #0,-(SP)    ; reference value
_NewWindow          ; make new window
```

-- ACTIVATE THIS NEW WINDOW -----

```
LEA WPOINTER,A0      ; copy window ptr
MOVE.L (SP),(A0)      ; to stacksave
_setport              ;current window
```

Now that we have a window on the desktop, we can draw into it. The remaining code sets up the event manager to detect a press of the mouse button. If the button has not been pressed, then we draw something (a box) in our window with a call to the subroutine QDSTUFF. Otherwise, when the button is pushed, we exit back to the finder. Note that the defined box we are drawing is set up as variables with the 'DS' assembly command. These values will be stored in the application globals area where they can be modified dynamically by our program if we wanted to animate the box.

--EVENT LOOP -----

```
MOVE.L #AllEvents,D0 ;all events
_FlushEvents          ;flushed
```

```
_InitCursor ; make cursor the arrow
```

GetEvent:

```
CLR -(SP)      ;returned event
MOVE #AllEvents,-(SP) ;mask all events
PEA EventRecord ; event record block
_GetNextEvent  ;go check the mouse
MOVE (SP)+,D0 ;get event result
CMP#0,D0       ;if 0 then no event
BEQGetEvent    ;loop until it happens
```

; JUMP TABLE OF EVENT PROCESSING

```
MOVE What,D0      ;what to do!
CMP#MouseDown,D0  ; button down?
BEQEXIT           ;yes so exit...
BSR QDSTUFF        ;no so draw box
JMP GetEvent       ;get next event
```

----- END OF MAIN -----

----- QDSTUFF SUBROUTINE -----

QDSTUFF:

```
LEA top,A0
MOVE.W #10, (A0) ;set up top
MOVE.W #30, 2(A0) ;left
MOVE.W #100, 4(A0) ;bottom
MOVE.W #200, 6(A0) ;right
```



```

PEA top      ;window rectangle
_FrameRect  ;draw rectangle
RTS

; --- RESTORE THE WORLD -----

EXIT:  LEA    SAVEREGS,A0    ; get 'em back
      MOVE.L  (A0),A6        ; local var
      MOVE.L  4(A0),A7       ; restore stack
      MOVEM.L (SP)+,D0-D7/A0-A6

; ---- RETURN TO FINDER -----

      RTS                ; return to finder

; ----LOCAL DATA AREA -----

SAVEREGS:  DCB.L  2,0        ;set save area
WPOINTER:  DC.L   0          ;store window pt
WBOUNDS:   DC.W   40         ;rectangle
          DC.W    2
          DC.W   335
          DC.W   508
WINDTITLE: DC.B   12         ; title length
          DC.B   'DAVES WINDOW',0

```

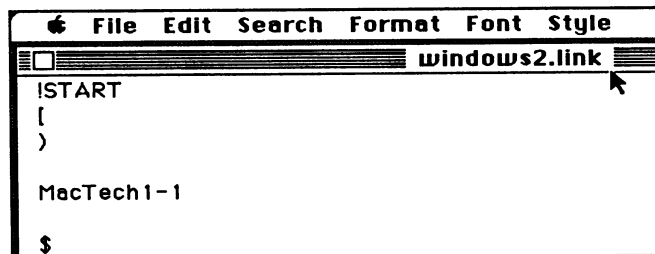


Fig. 6 The linker file for our asm program

```

EventRecord:

What:      DC.W  0          ; what event
Message:   DC.L  0          ; ptr. to msg
When:      DC.L  0
Point:     DC.L  0
Modify:    DC.W  0

```

```

EventTable:

DC.L      GetEvent      ;null event
DC.L      Exit          ;mouse down event
DC.L      GetEvent      ;mouse up
DC.L      GetEvent      ;key down event
DC.L      GetEvent      ;key up event
DC.L      GetEvent      ;auto key
DC.L      GetEvent      ;update event
DC.L      GetEvent      ;Disk Event
DC.L      GetEvent      ;activate event

```

; ----- APPLICATION GLOBALS -----

```

top:      DS.W          1
left:     DS.W          1
bottom:   DS.W          1
right:    DS.W          1

```

; ----- END OF PROGRAM -----

This completes our program. A typical Macintosh application follows this style of sitting in an event loop and waiting for something to happen. Join us next month for more from the Assembly Lab.

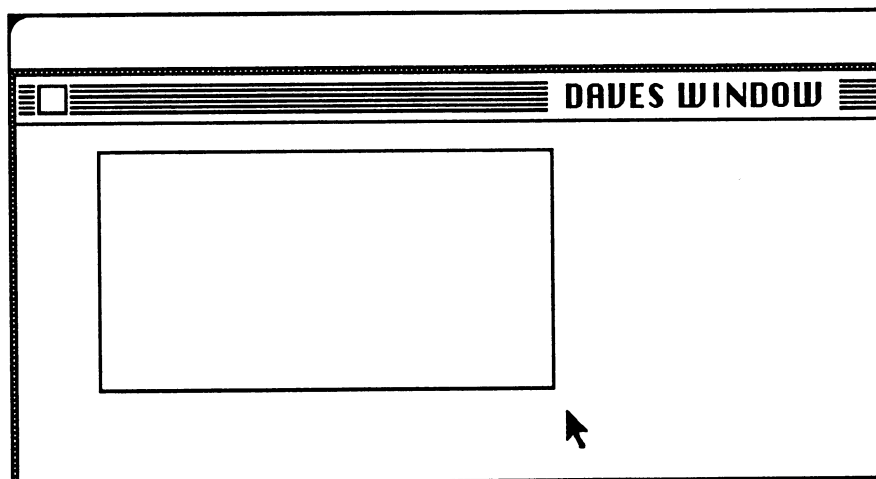
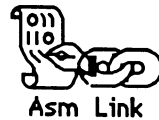


Fig. 7 Asm program output

Assembly Language Lab

Icons as Resources



David E. Smith
Editor & Publisher
MacTutor Vol. 1 No. 2

In last month's column, we presented a complete program shell for creating a Mac application from assembly language. We did not cover menus or menu bars, which we will save for next time. Also, we did not cover resources. This month, we begin examining resources, and in particular, icons, including how to install an icon on the desktop. If you've wanted to know how to create your own icons for your applications, then read on for a complete cookbook approach to this most unfamiliar of Mac topics.

Macintosh and Resources

Resources have to be the most frustrating aspect of learning how to program the Macintosh. It has been said the only reason they exist at all is to get around limitations of Lisa Pascal (a statement made by a programmer, no doubt). But they do serve a useful purpose once the program design is completed, in that they allow text-oriented program aspects like menus and dialogues to be separated from the code to facilitate translation into foreign languages. Apple is the first company to design into its product a universal market appeal. To date few developers have taken advantage of this feature, but it's there.

The main problem of resources facing new developers is that the concept is unique to the Macintosh, and almost totally undocumented. "Inside Macintosh", the only official Mac documentation from Apple, mentions only resource compilation on the Lisa using the Resource Compiler. It also mentions a "Resource Editor", but says it doesn't exist yet. What about the Mac environment? For the most part, that has to be dug out by hacking to translate Lisa resource formats into one of the Macintosh formats shown in figure 1. Both the Apple assembler/ linker and the RMaker utility can compile resources ON the Mac, FOR the Mac. The problem is, the formats are different than those described in the IM manuals for Lisa's Resource Compiler.

In addition to RMaker or the assembler system, other utilities are needed as shown in figure 2. Both icons and fonts are complicated enough to require their own editors for creating them. These utilities are available with the Software Supplement on Macintosh disks, or can be found in many public domain libraries. The remainder of this column will assume you have access to at least the icon editor and the Apple assembler/editor written by Bill Duvall of Consulair Corp.

Icons on the Desktop

The first thing most people want to do with resources is to create a unique icon on the desktop for their application

RESOURCE MAKERS

TOOL

COMPUTER

RMaker

Macintosh

ASM/Linker

Macintosh

Resource Compiler

Lisa

Resource Editor

Macintosh

fig. 1

program. Like a logo, an icon is often created before the first line of code is written! Creating an icon is easy. Getting it onto the desktop is not. The desktop file exhibits many peculiar properties when it comes to icons. If you move a file from one disk to another, you may be surprised to find the icon changes. This is because the Finder keeps a list of icons and the creator tags that go with each icon. If the desktop file already has an icon for a creator tag, it uses it, ignoring the icon resource in the application file. If it has never seen that creator tag before, it reads the file's resource to match the new icon with that creator type.

Desktop File Never Forgets!

The key to icons is the fact that the desktop file never forgets icon/creator tag combinations even if the file or application has been trashed. The only way to make it forget is to destroy the desktop file and force the Finder to re-create it from the resources in each application file. This is most easily done by booting up with the option/command key held down. This forces a new desktop file to be created, but throws everything out of the folders onto the desktop.

Creator Bytes Must Be Unique

If a file is moved to a disk where it never existed, but the creator tag has existed, the file will come up with the icon previously associated with that creator. This is why Apple requires icons and creator tags to be registered with them. The four byte creator tag must be unique or the whole Macintosh Icon scheme will be frustrated, as applications and documents lose track of each other. The TYPE and CREATOR tags tell the Finder which applications should be launched for which documents, and the icons are the visual indication of that linkage. Changing the creator tag of a resource file to

RESOURCE UTILITIES

TOOL

PURPOSE

- | | |
|-------------------|-------------------------------------|
| 1. RMaker | resource compiler on the Macintosh. |
| 2. Resource Mover | move resources between files. |
| 3. Set File | sets bundle bit, type and creator. |
| 4. Icon Edit | create icon files |
| 5. Font Editor | create new fonts. |
| 6. Font Mover | move fonts into the system file. |

fig. 2

something unique that the desktop file for that disk has never seen, will cause the new icon to appear without trashing the desktop file. Let's see how we go about creating new icons for our applications.

Icon Editor

The icon editor utility is used to create the new icon by using a "fat bits" mode to click in the icon form. A small window shows the actual icon proportions. An additional window displays the hex code format for the icon. Figure 3 shows the icon editor output as seen on the screen. The hex code for the icon, shown in figure 4, is saved to disk, with an appropriate name such as "WIND.ICON". The problem is, this binary information represents the graphic bit pattern of the icon and is incompatible with all our other resource tools. What is needed is a simple utility to read each byte of our icon file and re-write it in text format that either the assembler or RMaker can read.

Icon Converter Utility

The icon converter utility is written in MS/Basic 1.0 and is shown in the listing in figure 10. The output and set-up for the program is shown in figure 5. Icon Converter reads the icon file created by the icon editor and produces either a text

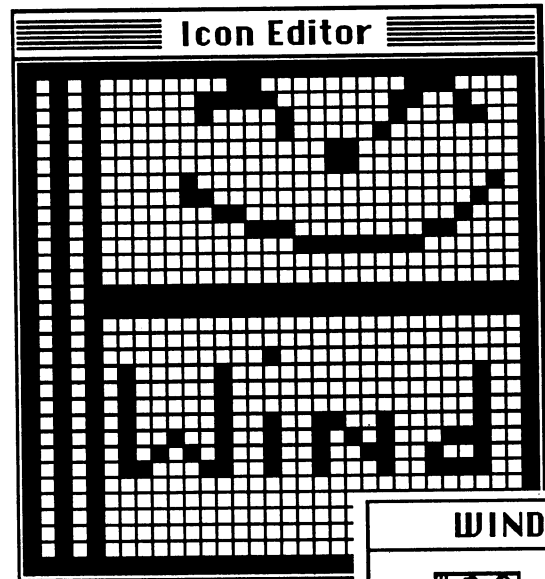


fig. 3a

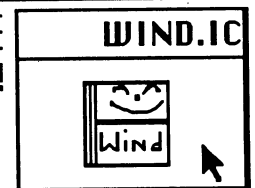


fig. 3b

file in assembly source code format for use by the assembler, or a text file in resource format for use by RMaker, the resource compiler on the Macintosh. As the questions in figure 5 indicate, the user can specify assembly or resource format for the output. Also, the program can generate an icon mask or read in a second icon file created by the icon editor for the mask.

Icon Mask

The icon mask is used when the icon is clicked. The mask icon is XOR'd with the icon to produce a black outline of the icon when it is selected. If the automatic mask option is selected with icon converter, then a mask of \$FFFF will be generated, which will result in the black outline shown for the icon in figure 6A. This produces a black square with the icon in white. To get the mask limited to the outline of the icon, a second icon must be created with the icon editor by filling an outline of the icon with solid black. This second icon can be

Hex Representation			
FFFFFFFF	A80600E1	A81F0191	A8108119
A8008201	A8001801	A8001801	A8200005
A8300009	A80C0011	A8038061	A8007F81
A8000001	A8000001	FFFFFFFF	FFFFFFFF
A8000001	A8000001	A8010001	AA080009
AA080009	AA080009	AA093209	AA492A39
AAA92649	AB192279	A8000001	A8000001
A8000001	A8000001	A8000001	FFFFFFFF

fig. 4

```

ICON CONVERTER.TXT
Name of Diskette (no colon) ? DAUE ASM 2
Name of File (CR for Finder) ? WIND4.ICON
ASM FILE OR RMAKER FILE (A or R) ? A
ADD MASK ICON OR GENERATE DEFAULT (A OR D) ? A
Name of Mask File ? WIND4.MASK
DC.L $00000000, $00020000, $00050000, $00080000
DC.L $00104000, $00222000, $00451000, $00880000
DC.L $01078400, $02204A00, $04402100, $08894000
DC.L $10508040, $20220020, $47140010, $88800008
DC.L $444003C4, $22500422, $114009D4, $08807808
DC.L $04000617, $020001A7, $01000047, $008000B7
DC.L $0040010F, $00200200, $00100400, $00080000
DC.L $00041000, $00022000, $00014000, $00008000

```

fig. 5

saved as "WIND.ICON.MASK" for example, and loaded by the icon converter by selecting the load mask option. The icon shown in figure 6B was created with a second icon for its mask. In either case, the program writes a text file of both the icon and its mask (either the automatic mask or the mask icon) to disk and names it with the ".ASM" or ".R" suffix, depending on if the assembler output or resource output is selected. The resulting output file can then be "INCLUDED" by the assembler in the assembly resource file, or merged with a resource file for compilation by RMaker.

Putting It All Together

Figure 7 shows an outline of the whole icon process. The assembler/linker system is the most reliable way to create icons and resources. The only problem is that the format of resources compiled with the assembler is not documented in the assembler manual. This format is different from the IM documentation for the Lisa resource compiler, as is the format

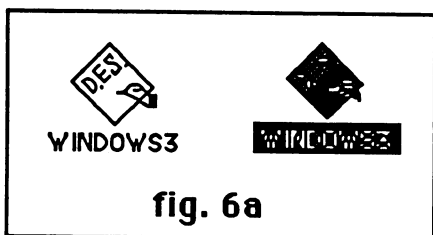


fig. 6a

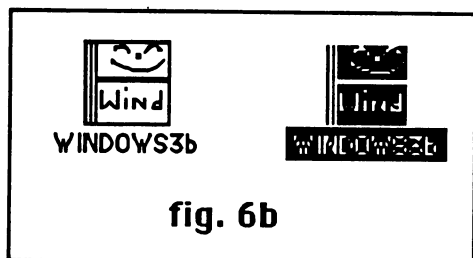
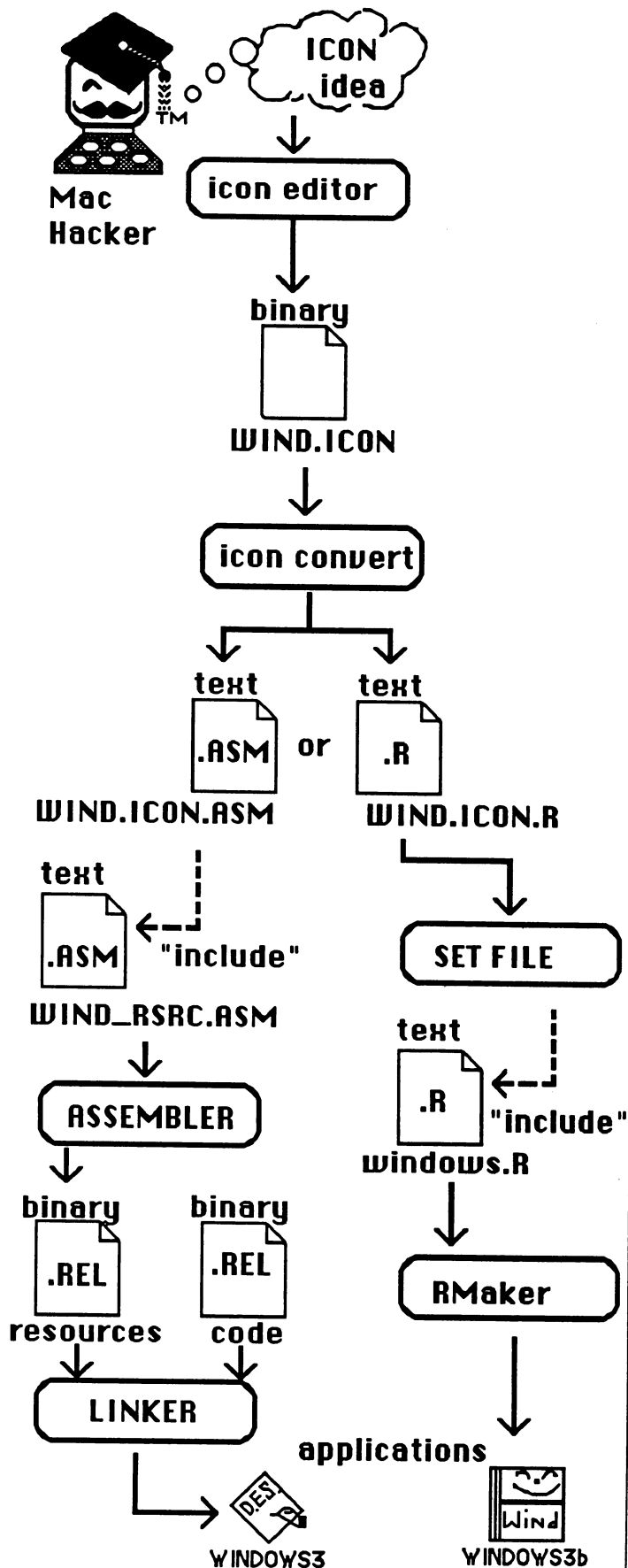


fig. 6b



for RMaker, which is only slightly better documented than the assembler format. After much trial and error, and an example file from Bill Duvall, the author of the assembler/linker, the correct assembly format for icons was found. The assembly source code is shown in figure 8. Icons that will be recognized by the Finder must include an ID string resource, a bundle resource, a file reference resource, and finally, the icon resource itself. Notice that all four of these resource types are shown in the assembly listing in figure 8.

Instead of copying the hex format for the icon from the icon editor, into the resource file, we simply use the INCLUDE function at the bottom of the file to read in the assembly source code for our icon created by the Icon Converter program. This produces the necessary "DC.L \$FF..." information for us. The resource file is compiled separately from the code file and then linked together using the Linker.

ID Tag Crucial

The key to getting the resource file to be recognized by the Finder, and the icon installed on the desktop, is to place the creator tag in several key places within the resource file. This includes the TYPE field in the identification string, the signature name in the bundle resource, and the creator field in the linker file. These three locations must contain the same four byte tag if the icon is to be loaded by the Finder. The first two locations are shown highlighted in figure 8. The creator tag is set in the linker file shown in figure 11. If RMaker is being used instead of the assembler, then both the id string and bundle tags can be set along with the creator tag within the resource file itself. Figure 9 shows the same resource file as figure 8, but in RMaker format instead of assembly format.

Assembler Versus RMaker

Which is easier to use, the assembler or RMaker, for creating resources? The assembler/linker is far more dependable and the files shown in this column are guaranteed to work. But the RMaker is a little better documented in the Apple assembler manual. After talking with Apple and Bill Duvall, it appears that Apple prefers to use RMaker for resources and the assembler for code, while Bill Duvall uses the assembler for both code and resources and never uses RMaker. Hence, resource creation on the Mac suffers from a dual personality. If the IM syntax were completely translated into Mac assembly syntax, then the assembler/linker method would be the most flexible and easiest to use.

A few words on using the Icon Converter program. The assembly and resource files created by this utility have neither a type or creator tag. This works fine with the assembler. The INCLUDE function is able to find and load the icon source code and assemble it along with the rest of the resources. The RMaker program, however, will not recognize the output of the Icon Converter program until the file's type and creator have been set to TEXT and EDIT by the Set File utility. In

many respects, RMaker is very much like a stripped-down Linker program. It produces an application file just as the linker does. Once the icon is created, code segments can be moved into the resource file to create a complete application. With the linker, code and resources are combined at link time. Both RMaker and the linker set the type, creator tags, and the bundle bit by using the new linker options "/BUNDLE", and "/TYPE" commands. The bundle bit tips off the Finder to check for an icon in the resource fork of the application. If the bundle bit is not set, no new icon!

The Linker File

Figure 11 shows the linker file used to link both the resources and the code file together to form a complete application. The code file is the same relocatable code we created last month for the windows program. It simply opens a window and draws a box on the screen. Due to space limitations, we have not re-printed the source listing for it here, but you can get that from last month's column. The important thing here is that any program code can be linked in place of our windows3 code, and the remaining linker commands will then link in the icon resource file to create a new icon on the desktop. Note the new linker commands for setting both the file type, creator and the bundle bit. To get this capability, you must have the final release version of the linker. Thanks for joining us in the Assembly Lab, and if you have something you've discovered, please write and share it with our readers.

Resource File In ASM Format

```
; WINDOWS3_RSRC.ASM
; resource file for windows3
; created using the assembler
; signature is creator tag
;
RESOURCE 'BOSW' 0 'IDENTIFICATION'

DC.B 32, 'WINDOWS3 MACTECH 1-2 - 23 NOV 84'

.ALIGN 2
RESOURCE 'BNDL' 128 'BUNDLE'

DC.L 'BOSW' ;NAME OF SIGNATURE
DC.W 0,1 ;DATA (DOESN'T CHANGE)
DC.L 'ICN#' ;ICON MAPPINGS
DC.W 0 ;NUMBER OF MAPPINGS-1
DC.W 0,128 ;MAP 0 TO ICON 128
DC.L 'FREF' ;FREF MAPPINGS
DC.W 0 ;NUMBER OF MAPPINGS-1
DC.W 0,128 ;MAP 0 TO FREF 128

RESOURCE 'FREF' 128 'FREF 1'

DC.B 'APPL', 0, 0, 0

.ALIGN 2
```

RESOURCE 'ICN#' 128 'MY ICON'

; FIRST APPLICATION ICON BIT MAP

INCLUDE WIND.ICON.ASM

fig. 8

RESOURCE FILE IN .R FORMAT

WINDOWS3c

APPLZSOB

*

* set up id string

*

TYPE ZSOB = GNRL

,0 ;; id name is signature

.l

10

.P

ABCDEFGHIJ

*

* set up file reference

*

TYPE FREF

,128 ;; resource id

APPL 0 ;; file type, id of icon

,129 ;; resource id

WIND 1 ;; file type, id of icon

*

* set up bundle resource

*

TYPE BNDL

,128 ;; resource id

ZSOB 0 ;; bundle owner

ICN# ;; bundled icons for appl

0 128 1 129 ;; local id 0 maps to 128

FREF

0 128 1 129 ;; appl ref 128, doc 129

*

* icon and shadow for application

*

TYPE ICN# = GNRL

,128

.H

FFFFFFFF A80600E1 A81F0191 A8108119

A8008201 A8001801 A8001801 A8200005

A8300009 A80C0011 A8038061 A8007F81

A8000001 A8000001 AFFFFFFF AFFFFFFF

A8000001 A8000001 A8010001 AA080009

AA080009 AA080009 AA093209 AA492A39

AAA92649 AB192279 A8000001 A8000001

A8000001 A8000001 A8000001 FFFFFFFF

FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF

FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF

FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF

FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF

Fig. 9

ICON CONVERTER

10 ' ICON CONVERTER

20 ' (C) DAVID SMITH MACTECH 1-2

30 '

40 DIM SEC\$(32),MEC\$(32),LIN\$(8)

50 CLS

60 CALL TEXTFONT(4)

70 INPUT "Name of Diskette (no colon) ";DSK\$

80 INPUT "Name of File (CR for Finder) ";FIL\$

90 '

100 IF FIL\$="" THEN SYSTEM

110 INPUT "ASM FILE OR RMAKER FILE (A or R)

";ASMFLG\$:ASMFLG=0:IF ASMFLG\$="A" THEN ASMFLG=1

120 INPUT "ADD MASK ICON OR GENERATE DEFAULT (A OR D) ";MSK\$

130 IF MSK\$="A" THEN MSK=1 ELSE MSK=0

140 IF MSK THEN INPUT "Name of Mask File ";FIL2\$

150 NAM\$=DSK\$+"." + FIL\$

160 NAM2\$=DSK\$+"."+FIL2\$

170 I=0:J=1:K=0:L=1

180 OPEN NAM\$ FOR INPUT AS #1

190 IF MSK THEN OPEN NAM2\$ FOR INPUT AS #2

200 '

210 WHILE NOT EOF(1)

220 KEY\$= INPUT\$(1,#1):IF MSK THEN KEY2\$=
INPUT\$(1,#2)

230 REC\$= HEX\$(ASC(KEY\$)):IF LEN(REC\$)=1 THEN
REC\$="0"+REC\$

240 IF MSK THEN MAS\$= HEX\$(ASC(KEY2\$)): IF
LEN(MAS\$)=1 THEN MAS\$="0"+MAS\$

250 IF MSK=0 THEN MAS=255:MAS\$=HEX\$(MAS): IF
LEN(MAS\$)=1 THEN MAS\$="0"+MAS\$

260 SEC\$(J)=SEC\$(J)+REC\$:MEC\$(J)=
MEC\$(J)+MAS\$:I=I+1: IF I=4 THEN J=J+1:I=0

270 WEND

280 '

290 ' END OF FILE FOUND

300 CLOSE #1:IF MSK THEN CLOSE #2

310 IF ASMFLG THEN OPEN NAM\$+".ASM" FOR OUTPUT AS
#1

320 IF ASMFLG=0 THEN OPEN NAM\$+".R" FOR
OUTPUT AS #1

330 '

340 ' COMPOSE TEXT STRINGS ICON

350 IF ASMFLG=0 THEN GOTO 480

360 '

370 ' ICON FOR ASSEMBLY FILE

380 FOR L=1 TO 8

390 LIN\$(L)="DC.L ":FOR J=1 TO 4: LIN\$(L)=
LIN\$(L)+"\$"+SEC\$(4*(L-1)+J)

400 IF J<4 THEN LIN\$(L)=LIN\$(L)+", "

410 NEXT J

420 NEXT L

430 '


```

440 FOR L=1 TO 8:PRINT LIN$(L):PRINT #1, LIN$(L):NEXT L:
    PRINT:PRINT#1, ""
450 GOTO 530
460 '
470 ' ICON FOR RESOURCE FILE
480 FOR L=1 TO 8:LIN$(L)="
490 FOR J=1 TO 4: LIN$(L)=
    LIN$(L)+SEC$(4*(L-1)+J)+" ": NEXT J
500 NEXT L
510 FOR L=1 TO 8:PRINT LIN$(L):PRINT #1, LIN$(L):NEXT L:
    PRINT:PRINT#1, ""
520 '
530 ' COMPOSE TEXT STRINGS FOR MASK ICON
540 IF ASMFLG=0 THEN GOTO 650
550 '
560 ' MASK FOR ASSEMBLY FILE
570 FOR L=1 TO 8
580 LIN$(L)="DC.L  ":FOR J=1 TO 4:
    LIN$(L)=LIN$(L)+"$"+ MEC$(4*(L-1)+J)
590 IF J<4 THEN LIN$(L)=LIN$(L)+", "
600 NEXT J
610 NEXT L
620 FOR L=1 TO 8:PRINT LIN$(L):PRINT #1, LIN$(L):NEXT L:
    PRINT:PRINT#1, ""
630 GOTO 710
640 '
650 ' MASK FOR RESOURCE FILE
660 FOR L=1 TO 8:LIN$(L)="
670 FOR J=1 TO 4: LIN$(L)=
    LIN$(L)+MEC$(4*(L-1)+J)+" ":NEXT J
680 NEXT L
690 FOR L=1 TO 8:PRINT LIN$(L):PRINT #1, LIN$(L):NEXT L:
    PRINT:PRINT#1, ""
700 '
710 ' INITIALIZE AND GO AGAIN
720 CLOSE #1
730 FOR I=0 TO 32: SEC$(I)="" :MEC$(I)="" :NEXT I
740 FOR I=1 TO 8:LIN$(I)="" :NEXT I
750 GOTO 80
760 END

```

LINKER FILE

```

!START
[
]
/OUTPUT WINDOWS3b

WINDOWS3

/TYPE 'APPL' 'BOSW'
/BUNDLE
/RESOURCES
WINDOWS3_RSRC

$

```

Fig. 11

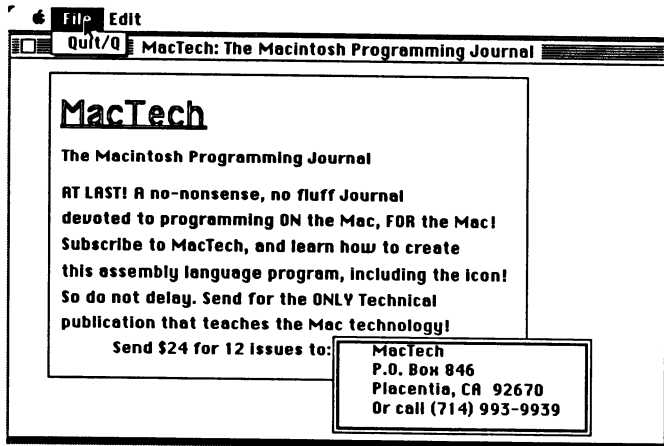


Assembly Language Lab

Double Clickable Start-up Ad



David E. Smith
Editor & Publisher
MacTutor Vol. 1 No. 3



Corvus called the other day and said they were sending 20 megabyte hard disks to various dealers and asked if I would like to put something about MacTutor on them. The problem was, they needed it the next day! So by 3 in the morning I had my double clickable start-up ad ready to express mail for the Corvus sampler. Thinking others might appreciate this little gem, I've used it as the basis for my column this month. It combines our application shell program from the December issue with our icon converter program from last month's issue, with some new material on menus and events.

What This Program Does

What we have here is a simple start-up program that can be set by the finder option as the program that gets executed when the disk first boots up. The program opens a window and draws an advertisement for MacTutor, as shown above in figure 1. When the close box is clicked, the finder is called. Such a program can be useful for the front end of your product development disk as a catchy way of introducing your product when the disk is booted up. You can modify the text to advertise your product, or make a clickable birthday card for your favorite Mac hacker. I gave a Mac Disk to a friend for Christmas and included this program with modified text to wish him a Merry Christmas everytime he started up his computer.

Toolbox Features

Our program may seem simple in operation, but it's already complex in toolbox features as listed below:

1. Drag window capability with window updating after drag.

2. Clickable close box exit.
3. Standard menu bar with apple, file and edit menus enabled.
4. Quit command in the file menu as an alternate return to the finder.
5. Text and box drawing using quickdraw.

Although the apple and edit pop-up menus are implemented, the support code for them is not here, so full support of the desk accessories will have to wait until next month. Everything required for a standard Macintosh program is here except text editing support, which is required for cut and paste. Only the quit option in the file menu actually does something. What we want to concentrate on here is the menu bar set-up. Then it is a minor problem to complete the stub routines to handle the individual apple and edit menu functions.

Menus as Resources

As we noted last month, resources suffer from almost no documentation and this includes menus as resources. Another problem is that there are several ways to create and maintain menus. They can be created from the program itself, or read in from a resource file. The resource file option can be either created using the RMaker, resource editor if it ever shows up, or by using the assembler. When creating menus from program code, the combination `_NewMenu`, `_AppendMenu`, and `_InsertMenu` traps are used. The `_AppendMenu` trap reads in the menu information from the local data area at the end of your program code, and the format of the menu items is well documented in the IM docs in the Menu Manager section. In fact, you can easily build your own RMaker using the append menu trap.

Append Menu Format

; This commented code shows how you
; would do menus using in line code
; instead of resources. Place this code
; in your menu set-up routine.

```
PEA APPLEMENU          ;menu set-up in code
_NewMenu                ;make new menu
MOVE.L (SP),D7          ;save menu handle
                        ;keep on stack
PEA APPLEMENUITEMS      ;apple menu items
_AppendMenu             ;add to apple menu
MOVE.L D7,-(SP)         ;menu handle
CLR.W -(SP)             ;add to end
_InsertMenu              ;insert menu items
```

; This commented code shows how you would

```
; set up resources in code for use with
; _AppendMenu instead of using a resource
; file with _GetRMenu. Place this section
; in your data block.
;
```

APPLEMENU:

```
DC.B 1
DC.B 20 ;apple symbol title
DC.B 0
```

APPLEMENUITEMS:

```
DC.B 21
DC.B 'ABOUT THIS PROGRAM...'
DC.B 0
```

```
; The remaining menu items would be
; desk accessories, which are added to
; this menu with another trap call.
```

Menu Items from Resources

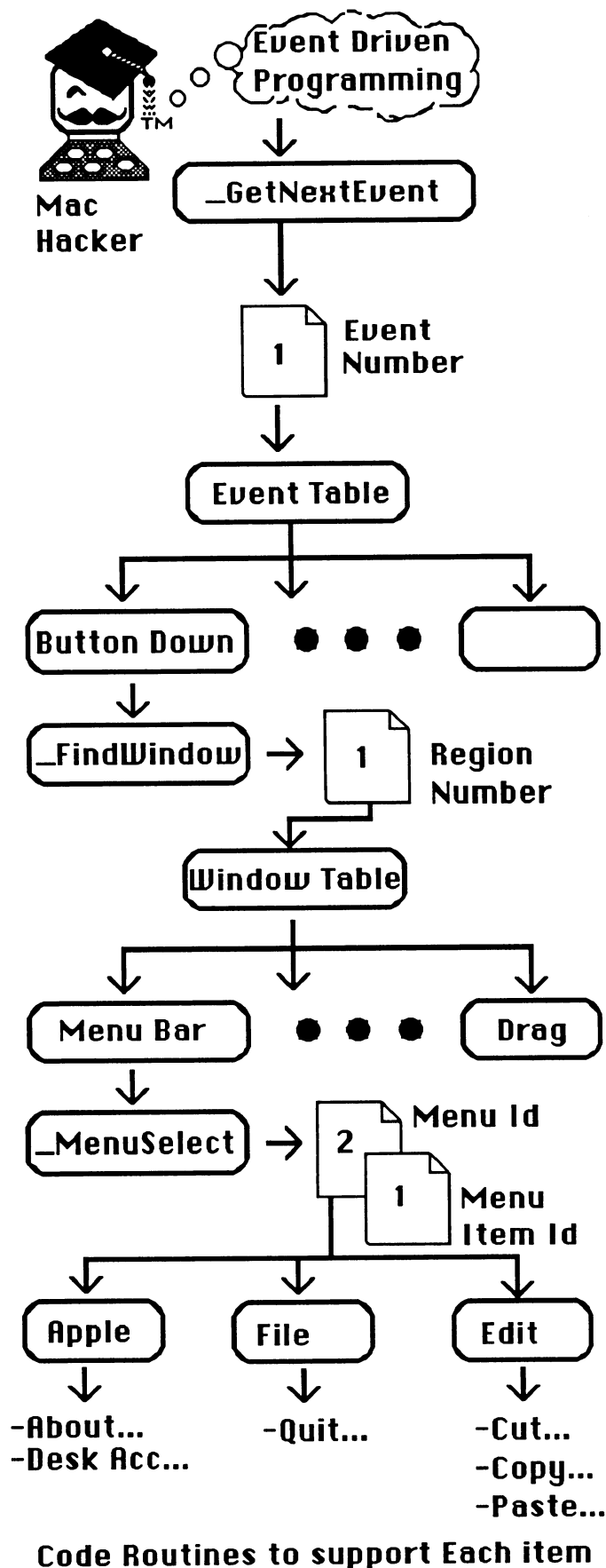
The reason for going through the above formats for using the append trap is because we DID NOT use it in our program! Instead, we used resources to define the menu and menu items and the _GetRMenu trap to read in the menu resource and set-up the menu bar. The resources can be created either by the RMaker utility, which is well documented, at least for making menus, or by the assembler, which is not documented at all and is not the same format as the RMaker input syntax. After much trial and error, I finally got the assembled resource file to work. It seems the assembler creates the binary format of each resource exactly as it is described in the IM docs. The trouble is, not all of the resources have their internal format documented. It is my guess that the RMaker program takes a high level description of the menu resources and translates that description into the same binary format. So in a sense, using the assembler meant we were our own RMaker!

Event Processing

The next major addition to this program is the event processing. Our event loop has been expanded from our simple application shell of the first issue. This time we use a jump table to process all the possible system events that could occur as a result of calling _SystemTask trap. This is done by using the event number returned from _GetNextEvent as an index into a table of offsets for handling each possible event. The big picture of how event processing works is shown in figure 2. In our program, the only event we do anything with is a mouse button down event. All the other entries in our event table return to the GetEvent loop.

Where's the mouse?

When the mouse button is pressed, and we branch by way of the event table to the ButtonDown routine, we first have to find out where the mouse is. This requires a SECOND table of offsets to branch to different routines depending on where the mouse was when the button was pressed. Our second table is called the WindowTable and it is indexed by the region number



Code Routines to support Each item

returned from `_FindWindow`, which has located the mouse for us. In this table, we have placed three routines; one for if the mouse is in the menu bar, another if the mouse is in the go-away region of the window, and finally, a third if the mouse is in the drag region of the window.

The drag region routine is easy. We just call `_DragWindow` trap to move the window. But we also have to re-draw our ad or some of it will be erased when the window is moved. Fortunately all of our ad is drawn in a single subroutine called `QDSTUFF`, which can be called at any time, including after a drag window operation. And so as if by magic, the ad is restored.

The go-away region is also simple to handle. We just exit our program and return to the finder. The hard task is the menu bar region. Now we have to find out which menu we are in and go through another set of branches to handle each menu item.

Menu Bar Processing

We now come to our third and fourth levels of event processing. The third level is our routine `Menubar`, which we branch to when we find out the mouse is in the menu bar. Now we have to find out which menu the mouse is in. This is done by calling `_MenuSelect` trap. The returned menu Id and menu item Id identify the menu and the associated menu item where the mouse was clicked. Since we have three menus, Apple, File and Edit, we have three more branches to a routine for each menu. At this point, only the File menu item is processed. We check if the quit option was selected and exit the program. If we were in the apple menu, we would need to make a fourth level branch to a routine for handling either the About option or the desk accessories. This code we have not added yet. Instead `Inapplemenu` and `Ineditmenu` just return to the first level of our event processing.

What About the Ad?

We've spent so much time just checking out the mouse, we nearly forgot about our ad! The `QDSTUFF` subroutine draws our ad in our window in the quickest, most brute force way. We simply call `_DrawString` from `quickdraw` to draw strings in our window to describe our product. A few boxes live it up. The nice thing about `_DrawString` is that it doesn't require character counting! Just specify the location and away it goes. A few different fonts also help.

Make a Good Icon!

Any advertisement needs a good icon and ours is shown on the column masthead. We learned how to create this icon on the desktop in our last issue. To review, we create the icon using the icon editor supplied by Apple. Then we run the icon binary file through our Icon Converter program to create assembly source code. Our resource file has an Include statement that will then merge in this assembly source file for the icon into our resource file. When we assemble the

resource file and link it with the program code, we get a nice icon on the desktop. Refer to last month's issue for the Icon Converter program source code.

Next time we will complete our event processing with the About dialog box, desk accessory support and the remaining support code for our edit menu items. Have some good ideas on assembly programming? Send them to MacTutor and share them with others!

```
; EXAMPLE ASSEMBLY PROGRAM
; MENUBAR (MacTutor 1-3)
; VERSION 26 DEC 84
; (C) 1984 MacTec by David E. Smith
```

```
; Macro subset for Toolbox stuff
```

```
MACRO _InitGraf = DC.W $A86E|
MACRO _InitWind =          DC.W $A912|
MACRO _NewWindow =         DC.W $A913|
MACRO _setport =           DC.W $A873|
MACRO _InitFont =          DC.W $A8FE|
MACRO _InitMenu =          DC.W $A930|
MACRO _InitDialog =        DC.W $A97B|
MACRO _TEInit =             DC.W $A9CC|
MACRO _Initpack =           DC.W $A9E5|
MACRO _FlushEvents =        DC.W $A032|
MACRO _InitCursor =         DC.W $A850|
MACRO _GetNextEvent =       DC.W $A970|
MACRO _FrameRect =          DC.W $A8A1|
MACRO _TextFont =           DC.W $A887|
MACRO _TextFace =           DC.W $A888|
MACRO _TextSize =           DC.W $A88A|
MACRO _MoveTo =             DC.W $A893|
```

```
MACRO _DrawString =         DC.W $A884|
MACRO _PenSize =            DC.W $A89B|
MACRO _EraseRect =          DC.W $A8A3|
MACRO _GetRMenu =           DC.W $A9BF|
MACRO _InsertMenu =         DC.W $A935|
MACRO _DrawMenuBar =        DC.W $A937|
MACRO _Debugger =           DC.W $A9FF|
MACRO _NewMenu =            DC.W $A931|
MACRO _AppendMenu =         DC.W $A933|
MACRO _AddResMenu =         DC.W $A94D|
MACRO _FindWindow =         DC.W $A92C|
MACRO _DragWindow =         DC.W $A925|
MACRO _SystemTask =         DC.W $A9B4|
MACRO _MenuSelect =         DC.W $A93D|
MACRO _HiLiteMenu =         DC.W $A938|
```

```
; DECLARE LABELS EXTERNAL
```

```
XDEF START ; required for linker
```

```
; LOCAL EQUATES
```

```
MouseDown equ 1
MaxEvents equ 12
AllEvents equ $0000FFFF
```

```
; MAIN PROGRAM SEGMENT
```

```

DC.W 'MINE' ;find start of program

; --- SAVE THE WORLD -----

START: MOVEM.L D0-D7/A0-A6, -(SP)
      LEA  SAVEREGS,A0
      MOVE.L  A6,(A0) ; local var
      MOVE.L  A7,4(A0) ; stack ptr

; ---INITIALIZE ALL MANAGERS-----

; SET UP QUICKDRAW GLOBALS

PEA -4(A5) ;push qd global ptr
_InitGraf ;init quickdraw global

;---- SET UP REMAINING MANAGERS ---

_InitFont ; init font manager
_InitWind ; init window manager
_InitMenu ; init menu manager

CLR.L -(SP) ; kill the restart
_InitDialog ; init dialog manager

_TEInit ; init text edit (ROM)

MOVE.W #2,-(SP) ; set-up
_Initpack ; init package mgr

MOVE.L #AllEvents,D0 ;all events
_FlushEvents ;flushed

_InitCursor ; make cursor the arrow

; SET UP MENU BAR WITH POP-UP MENUS

CLR.L -(SP) ;returned menu handle
MOVE.W #1,-(SP) ;push menu ID 1 Apple
_GetRMenu ;get menu from resource
LEA APPLEMENU,A0 ; copy apple menu handle
MOVE.L (SP),(A0) ; to local data block
MOVE.L (SP), -(SP) ;menu handle desk acc.
;menu handle on stack

CLR.W -(SP) ;push 0 for append
_InsertMenu ;add menu item

MOVE.L #'DRVr', -(SP) ;load all DRVr type
_AddResMenu ;add desk accessories

CLR.L -(SP) ;return menu handle
MOVE.W #2,-(SP) ;push menu ID 2 File
_GetRMenu ;get menu from resource
;leave handle on stack

CLR.W -(SP) ;append to menu
_InsertMenu ;the File stuff

CLR.L -(SP) ;return menu handle

```

```

MOVE.W #3,-(SP) ;push menu ID 3 Edit
_GetRMenu ;get menu from resource
;leave handle on stack

CLR.W -(SP) ;append to menu
_InsertMenu ;the File stuff

_DrawMenuBar ;our menu bar!

```

;- SET UP NEW HEAP WINDOW ---

```

CLR.L -(SP) ;return window ptr
CLR.L -(SP) ;window record ptr.
PEA WBOUNDS ;window rectangle
PEA WINDTITLE ; window title
MOVE.W #$100,-(SP) ; true = visible
MOVE.W #0,-(SP) ; doc type window
MOVE.L #-1,-(SP) ; window in front
MOVE.W #$100,-(SP) ; true=closebox
MOVE.L #0, -(SP) ; reference value
_NewWindow ; make new window

```

;- ACTIVATE THIS NEW WINDOW -----

```

LEA WPOINTER,A0 ; copy window ptr
MOVE.L (SP),(A0) ; to stacksave
_setport ;current window

```

; --- SET UP WINDOW DISPLAY -----

```

BSR QDSTUFF ;draw text with QD

```

; ----- EVENT LOOP -----

GetEvent:

```

_SystemTask ;check out desk acc.
CLR -(SP) ;returned event
MOVE #AllEvents,-(SP) ;mask all events
PEA EventRecord ; event record block
_GetNextEvent ;go check the mouse
MOVE (SP)+,D0 ;get event result
CMP #0,D0 ;if 0 then no event
BEQ GetEvent ;loop until it happens

```

; JUMP TABLE OF EVENT PROCESSING

```

CLR.L D0
MOVE What,D0 ;what to do! (event number)
CMP #MaxEvents,D0 ;event no. >11?
BGE GetEvent ;yes so ignore it
ADD D0,D0 ;multiply by 2
MOVE EventTable(D0),D0
JMP EventTable(D0) ;execute event

```

EventTable:

```

DC.W GetEvent-EventTable ;null event
DC.W ButtonDown-EventTable ;mouse down
DC.W GetEvent-EventTable ;mouse up
DC.W GetEvent-EventTable ;key down event
DC.W GetEvent-EventTable ;key up event
DC.W GetEvent-EventTable ;auto key

```

```

DC.W      GetEvent-EventTable ;update event
DC.W      GetEvent-EventTable ;Disk Event
DC.W      GetEvent-EventTable ;activate event
DC.W      GetEvent-EventTable ;Abort
DC.W      GetEvent-EventTable ;Network
DC.W      GetEvent-EventTable ;I/O Driver
           ;application events would
           follow

```

; note that linker error may result if
; the table entry is >128

; - EVENT PROCESSING ROUTINES -----

ButtonDown:

```

CLR      -(SP)      ;result of findwindow
MOVE.L   Point, -(SP) ;push mouse coordinates
PEA      WWindow    ;push holder for handle
_FindWindow      ;where was mouse click?
CLR.L    D0
MOVE (SP)+,D0      ;pop region number
ADD      D0,D0      ;multiply by 2
MOVE     WindowTable(D0),D0
JMP      WindowTable(D0) ;do mouse down event

```

WindowTable: ;table of mouse events

```

DC.W      GetEvent-WindowTable
           ;in desk (none of following)
DC.W      Menubar-WindowTable
           ;in menu bar
DC.W      GetEvent-WindowTable
           ;in system window
DC.W      GetEvent-WindowTable
           ;in content region
DC.W      Drag-WindowTable
           ;in drag region
DC.W      GetEvent-WindowTable
           ;in grow region
DC.W      Exit-WindowTable
           ;in go-away region

```

Drag: ;mouse in drag window

```

MOVE.L   WWindow, -(SP) ;push window pointer
MOVE.L   Point, -(SP) ;push mouse coordinates
PEA      DRAGWINDOW    ;window boundry to drag
_DragWindow      ;move window
BSR      QDSTUFF        ;update window text
JMP      GetEvent      ;return to event loop

```

Menubar:

```

CLR.L    -(SP)      ;returned menu choice
MOVE.L   Point, -(SP) ;mouse position
_MenuSelect      ;Find menu selected
MOVE     (SP)+,D5    ;pop menu ID to D5
MOVE     (SP)+,D6    ;pop menu item ID to D6
CLR      -(SP)      ;select all menus
_HiLiteMenu      ;unhilite them all

```

```

CMP      #1, D5      ;apple menu?
BEQ      Inapplemenu
CMP      #2, D5      ;file menu?
BEQ      Infilemenu
CMP      #3,D5       ;edit menu?
BEQ      Ineditmenu
BRA      GetEvent    ;no place else

```

Inapplemenu:

```
BRA      GetEvent    ;no place else
```

Infilemenu:

```

CMP      #1,D6       ;quit?
BNE      GetEvent    ;no, try again
BRA      Exit        ;yes, exit

```

Ineditmenu:

```
BRA      GetEvent    ;no place else
```

Exit:

```
JMP      Tofinder    ;return to finder
```

; -- END OF MAIN -----

; -- QDSTUFF SUBROUTINE -----

QDSTUFF:

; This subroutine is a static
; window display of text for our
; ad. It would more properly be
; placed in a resource file as
; strings, but when in a hurry...

```

LEA      top,A0      ;set-up our globals
           ;window frame size.
MOVE.W   #10, (A0)    ;TOP
MOVE.W   #30, 2(A0)   ;LEFT
MOVE.W   #250, 4(A0)  ;BOTTOM
MOVE.W   #400, 6(A0)  ;RIGHT

PEA      top          ;window rectangle
_FrameRect      ;draw rectangle

MOVE.W   #1, -(SP)    ;FONT
_TextFont
MOVE.W   #12, -(SP)   ;style
_TextFace
MOVE.W   #24, -(SP)   ;SIZE
_TextSize
MOVE.W   #40, -(SP)   ; h
MOVE.W   #50, -(SP)   ;v
_MoveTo      ;set pen location
PEA      'MacTutor'
_DrawString

MOVE.W   #0, -(SP)    ;chicago

```

```

_TextFont
MOVE.W #0, -(SP) ;normal
_TextFace
MOVE.W #12, -(SP) ;12 point size
_TextSize
MOVE.W #40, -(SP) ;H
MOVE.W #80, -(SP) ;V
_MoveTo
PEA 'The Macintosh Programming Journal'
_DrawString

MOVE.W #40, -(SP) ;H
MOVE.W #110, -(SP) ;V
_MoveTo
PEA 'AT LAST! A no-nonsense, no fluff Journal'
_DrawString

MOVE.W #40, -(SP) ;H
MOVE.W #130, -(SP) ;V
_MoveTo
PEA 'devoted to programming ON the Mac, FOR the Mac!'
_DrawString

MOVE.W #40, -(SP) ;H
MOVE.W #150, -(SP) ;V
_MoveTo
PEA 'Subscribe to MacTutor, and learn how to create'
_DrawString

MOVE.W #40, -(SP) ;H
MOVE.W #170, -(SP) ;V
_MoveTo
PEA 'this assembly language program, including the icon!'
_DrawString

MOVE.W #40, -(SP) ;H
MOVE.W #190, -(SP) ;V
_MoveTo
PEA 'So do not delay. Send for the ONLY Technical '
_DrawString

MOVE.W #40, -(SP) ;H
MOVE.W #210, -(SP) ;V
_MoveTo
PEA 'publication that teaches the Mac technology!'
_DrawString

MOVE.W #80, -(SP) ;H
MOVE.W #230, -(SP) ;V
_MoveTo
PEA 'Send $24 for 12 issues to:'
_DrawString

LEA top,A0
MOVE.W #215, (A0) ;TOP
MOVE.W #250, 2(A0) ;LEFT
MOVE.W #290, 4(A0) ;BOTTOM
MOVE.W #450, 6(A0) ;RIGHT
PEA top ;window rectangle
_EraseRect ;clear rectangle
PEA top
_FrameRect ;draw rectangle

```

```

MOVE.W #2, -(SP) ;WIDTH
MOVE.W #2, -(SP) ;HEIGHT
_PenSize

LEA top,A0
MOVE.W #218, (A0) ;TOP
MOVE.W #253, 2(A0) ;LEFT
MOVE.W #287, 4(A0) ;BOTTOM
MOVE.W #447, 6(A0) ;RIGHT
PEA top ;window rectangle
_FrameRect ;draw rectangle

MOVE.W #280, -(SP) ;H
MOVE.W #230, -(SP) ;V
_MoveTo
PEA 'MacTutor'
_DrawString

MOVE.W #280, -(SP) ;H
MOVE.W #245, -(SP) ;V
_MoveTo
PEA 'P.O. Box 400'
_DrawString

MOVE.W #280, -(SP) ;H
MOVE.W #260, -(SP) ;V
_MoveTo
PEA 'Placentia, CA 92670'
_DrawString

MOVE.W #280, -(SP) ;H
MOVE.W #275, -(SP) ;V
_MoveTo
PEA 'Or call (714) 630-3730'
_DrawString

RTS

; ----- MOVBALL SUBROUTINE -----

MOVBALL:
; place some animation in this routine
; while waiting for an event.
RTS

; --- RESTORE THE WORLD -----

Tofinder: LEA SAVEREGS,A0 ; get back
MOVE.L (A0),A6 ; local var
MOVE.L 4(A0),A7 ;restore stack
MOVEM.L (SP)+,D0-D7/A0-A6

; --- RETURN TO FINDER -----

RTS ; return to finder

; ---LOCAL DATA AREA -----

```



```

APPLEMENU:  DC.L  0

SAVEREGS:   DCB.L  2,0  ;set save area

WPOINTER:   DC.L  0  ;store window pt

WBOUNDS:    DC.W  40  ;rectangle TOP
             DC.W  2  ;LEFT
             DC.W  335;BOTTOM
             DC.W  508;RIGHT

WINDTITLE:  DC.B  42  ;title length
             DC.B  'MacTutor: The Macintosh Programming
Journal',0

DRAGWINDOW: DC.W  30  ;RECTANGLE
             DC.W  1
             DC.W  350
             DC.W  510

EventRecord:

  What:      DC.W  0  ;what event number
  Message:   DC.L  0  ;ptr. to msg
  When:      DC.L  0  ;Time event posted
  Point:     DC.L  0  ;mouse coordinates
  Modify:    DC.W  0  ;state of keys
  WWindow:   DC.L  0  ;Find window's result

;--  APPLICATION GLOBALS  -----

top:        DS.W  1
left:       DS.W  1
bottom:     DS.W  1
right:      DS.W  1

;----  END OF PROGRAM  -----

; MENUBAR_RSRC.ASM
; resource file for the MENUBAR
; created using the assembler
; signature is creator tag
;
RESOURCE 'DAVE' 0 'IDENTIFICATION'

      DC.B  18, 'AD FOR MACTECH 1-3'

.ALIGN 2
RESOURCE 'BNDL' 128 'BUNDLE'

      DC.L  'DAVE'      ;NAME OF SIGNATURE
      DC.W  0,1         ;DATA (DOESN'T CHANGE)
      DC.L  'ICN#'      ;ICON MAPPINGS
      DC.W  0           ;NUMBER OF MAPPINGS-1
      DC.W  0,128       ;MAP 0 TO ICON 128

      DC.L  'FREF'      ;FREF MAPPINGS
      DC.W  0           ;NUMBER OF MAPPINGS-1
      DC.W  0,128       ;MAP 0 TO FREF 128

```

RESOURCE 'FREF' 128 'FREF 1'

```
DC.B  'APPL', 0, 0, 0
```

```
.ALIGN 2
```

RESOURCE 'ICN#' 128 'MY ICON'

```
; FIRST APPLICATION ICON BIT MAP
```

INCLUDE LOGO.ICON.ASM

```
; MENU BAR RESOURCES
```

```
.ALIGN 2
```

RESOURCE 'MENU' 1 'apple menu'

```

DC.W 1      ;MENU ID
DC.W 0      ;WIDTH HOLDER
DC.W 0      ;HEIGHT HOLDER
DC.L 0      ;RESOURCE ID HOLDER FOR STD. MENU
DC.L $1FF   ;ENABLE ALL ITEMS
DC.B 1      ;TITLE LENGTH
DC.B 20     ;APPLE SYMBOL

```

```

DC.B 21     ;MENU ITEM LENGTH
DC.B 'ABOUT THIS PROGRAM...'
DC.B 0      ;NO ICON
DC.B 0      ;NO KEYBOARD EQUIVALENT
DC.B 0      ;MARKING CHARACTER
DC.B 0      ;STYLE OF ITEM'S TEXT
DC.B 0      ;END OF MENU ITEM

```

```
.ALIGN 2
```

RESOURCE 'MENU' 2 'file menu'

```

DC.W 2      ;MENU ID
DC.W 0      ;WIDTH HOLDER
DC.W 0      ;HEIGHT HOLDER
DC.L 0      ;RESOURCE ID HOLDER
DC.L 3      ;ENABLE ALL ITEMS
DC.B 4      ;TITLE LENGTH
DC.B 'File' ;file menu

```

```

DC.B 6      ;MENU ITEM LENGTH
DC.B 'Quit/Q'
DC.B 0      ;NO ICON
DC.B 0      ;NO KEYBOARD EQUIVALENT
DC.B 0      ;NO MARKING CHARACTER
DC.B 0      ;STYLE OF ITEM'S TEXT
DC.B 0      ;END OF MENU ITEM

```

```
.ALIGN 2
```

RESOURCE 'MENU' 3 'edit menu'

```

DC.W 3      ;MENU ID
DC.W 0      ;WIDTH HOLDER
DC.W 0      ;HEIGHT HOLDER
DC.L 0      ;RESOURCE ID HOLDER
DC.L $7B    ;ENABLE ALL ITEMS except 2
DC.B 4      ;TITLE LENGTH
DC.B 'Edit' ;edit menu

```

DC.B 6 ;MENU ITEM LENGTH
DC.B 'Undo/Z'
DC.B 0 ; NO ICON
DC.B 0 ; NO KEYBOARD EQUIVALENT
DC.B 0 ; NO MARKING CHARACTER
DC.B 0 ; STYLE OF ITEM'S TEXT

DC.B 9 ;MENU ITEM LENGTH
DC.B '-----'
DC.B 0 ; NO ICON
DC.B 0 ; NO KEYBOARD EQUIVALENT
DC.B 0 ; NO MARKING CHARACTER
DC.B 0 ; STYLE OF ITEM'S TEXT

DC.B 5 ;MENU ITEM LENGTH
DC.B 'Cut/X'
DC.B 0 ; NO ICON
DC.B 0 ; NO KEYBOARD EQUIVALENT
DC.B 0 ; NO MARKING CHARACTER
DC.B 0 ; STYLE OF ITEM'S TEXT

DC.B 6 ;MENU ITEM LENGTH
DC.B 'Copy/C'
DC.B 0 ; NO ICON
DC.B 0 ; NO KEYBOARD EQUIVALENT
DC.B 0 ; NO MARKING CHARACTER
DC.B 0 ; STYLE OF ITEM'S TEXT

DC.B 7 ;MENU ITEM LENGTH
DC.B 'Paste/V'
DC.B 0 ; NO ICON

DC.B 0 ; NO KEYBOARD EQUIVALENT
DC.B 0 ; NO MARKING CHARACTER
DC.B 0 ; STYLE OF ITEM'S TEXT

DC.B 5 ;MENU ITEM LENGTH
DC.B 'Clear'
DC.B 0 ; NO ICON
DC.B 0 ; NO KEYBOARD EQUIVALENT
DC.B 0 ; NO MARKING CHARACTER
DC.B 0 ; STYLE OF ITEM'S TEXT

DC.B 0 ; END OF MENU ITEMS

```
!START
[
]
)
/OUTPUT Menubar start-up

Menubar

/TYPE 'APPL' 'DAVE'
/BUNDLE
/RESOURCES
Menubar_rscs

$
```

Fig. 3 Linker File



Assembly Language Lab

Modeless Dialogs and DA Support



David E. Smith
Editor & Publisher
MacTutor Vol. 1 No. 4

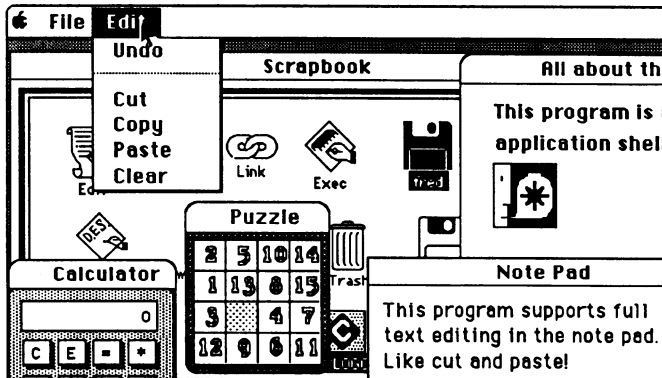


Fig. 1 "Boxes" Program supports Desk Accessories

Last month we covered menu bars and event loops, but we left out complete desk accessory support. Figure 1 above shows that we've completed desk accessory support in this month's "Boxes" program. Full cut and paste is supported in the note pad, and we have a dialog box to tell about our program. This represents a complete Macintosh application shell program for any game-type application that does not need text editing within the application window, but wants a standard user interface. Because of the length of the program, we will have to keep our comments brief and try to expand on them in coming issues.

Notice that the number of trap calls needed to fully support the standard application interface has increased dramatically since last month. Even a simple program, if it uses all the Mac features, will touch nearly every ROM manager. This month's program adds 27 ROM calls to the listing we presented last month.

Text Edit Support

The key to full desk accessory support is text editing using cut and paste. Once this is successfully implemented, the rest is easy. The TextEdit manager handles all text display and management including the actual cutting and pasting. We don't have to do hardly anything except provide a text record for him. This is done with `_TENew` trap. We clear room for the returning handle to the text record, push two rectangles and call the trap to set-up a text edit record on the heap. The rectangles control the text destination on the screen and the clipping of that text to a display rectangle. See the "Set Up Text Edit" section of our program to see how this is done.

As Bob Denny mentioned in his "C" column, the text edit manager uses our EDIT menu, so we have to set-up our EDIT

menu items in the standard order. Then, the actual text editing in a desk accessory is done for us by the trap `_SysEdit`. We check this trap first if we find the mouse in the edit menu to allow cutting and pasting by a desk accessory. The only support we need is for each of our EDIT menu item routines to call the corresponding Text Edit manager call that does the cutting, pasting or clearing on our text. We push the handle to our text record obtained from `_TENew`, and let the ROM routines do it!

Modeless Dialog Boxes

Most applications have a "Modal" dialog box that serves as the "About" function to tell the user about the program. This is the first item in the Apple menu. We've implemented a "Modeless" dialog box here. What that means is the user is free to make use of other desk accessories or the application itself while the "About" box is being displayed. This opens up the possibility of the user again selecting the "About" box, which would cause TWO dialog boxes to be displayed! Figure 2 shows how our program allows any number of dialog boxes to be created by the user. This is interesting because it lets us slide into multiple window support without really doing multiple windows. Each dialog box is a small window set-up on the heap. The Dialog manager takes care of most of the upkeep for this mini-window. Notice our dialog box also has two icons in it as well as some static text. The resource file shows how the icons and static text are set-up in assembler format so they will be displayed properly in the "About" box.

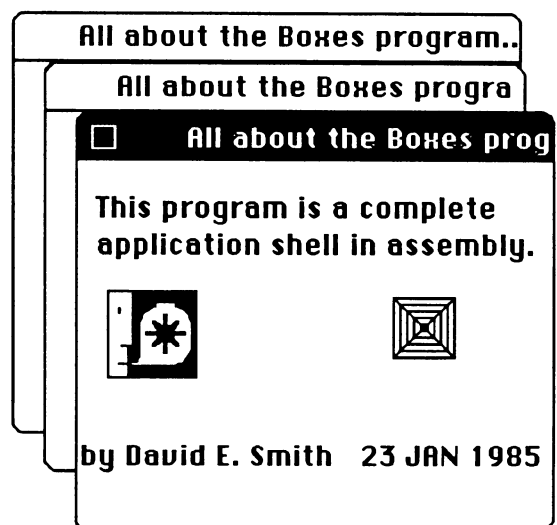


Fig. 2 "About" Modeless Dialog Box

Dialog Box Event Support

We have expanded our event loop from last month to include support for the dialog box. Supposedly the dialog traps check for a dialog box event (`_IsDialogEvent`), and process such an event for us (`_DialogSelect`), allowing us to be unconcerned about events affecting the "About" window such as dragging or closing. In actuality, the dialog routine for executing dialog events, `_DialogSelect` is quite limited and checks must be made in the application event loop to see if some action is required for a dialog box. Notice in our event loop that we check for a dialog event right after calling `_GetNextEvent` and before branching to our standard event routines. Dragging the dialog window or clicking the go away box in the dialog window are not indicated by the `_DialogSelect` routine. Apparently only a click on a control within the dialog box is returned as a "true" result.

Our Event Loop

This month our event loop is expanded to handle activate and deactivate events. We've added an OPEN and CLOSE feature to our application FILE menu to allow the user to selectively open and close the main window for our program. When the window is open and active, we disable the edit menu items since our application does not support these. We must then enable them again if our window is deactivated so any desk accessories can use them. This is done in the activate and deactivate event routines. The message and modify sections of our event record tell us if the activate was for our window, and whether we should activate or deactivate.

Update Event

The hardest part about our program was getting the update event to work properly. If our window has been dragged, we will get an update event to re-draw it. I was having trouble because when I called `QDSTUFF` to re-draw the window, everything was going into the dialog box! This was solved by re-setting the current graf port to our window so that an update calling to `QDSTUFF` would be sure to draw in our window and not the dialog box, if it was still hanging around. An update event for a dialog window is handled by the dialog manager. However a drag event could be a dialog window. Since our program allows multiple dialog windows, if any of them are dragged, we re-store the window location so any new dialog windows don't run off the screen. Until a window is dragged, multiple dialog windows appear down and right of the previous one as shown in figure 2. The "About" routine takes care of this when we create a new dialog box by incrementing the window location box.

Go Away Box

Another tricky part to dialog boxes was when the "go away" box was clicked. It seems that `_CloseDialog` will hang the system if used for modeless dialog boxes where more than one such box is allocated. So `_CloseWindow` should be used

instead so that the support structure for the remaining dialog windows is not deleted. In our "HIDE" routine, if the "go away" box is clicked, we check if it's a dialog window or our application window. If it's the dialog window, we update the window location by backing it up so that the next dialog window will continue to appear down and to the right of the previous one. However, if we close all of them back to the original starting location for our chain of dialog windows, we re-set everything so the dialog windows do not appear to be going backwards! Many of these little concerns will also affect using multiple windows and so are important to take care of here where things are still easy.

Desk Accessory Support

Once the text edit record is set up and the EDIT menu items in place, little is needed to support desk accessories. We find the desk accessory with `_GetItem`, and then push the accessory's name and call `_OpenDeskAcc` to do it. We also check `_SysEdit` in our EDIT menu item routine as mentioned to allow any open desk accessory to do text editing.

The "About" function reads the dialog item list from the resource file and opens a dialog window on the heap in much the same manner as `_NewWindow` opens a window on the heap. The "DITL" resource type holds the item list of those controls, text, icons or whatever that will appear in the dialog box. In our case, we have only static text and two icons in our "DITL" resource. The first icon is a system icon that displays a note. The second is one of ours that displays the logo for this program. Note that for this to work, the "ICON" resource must be used, not the icon list resource, as I found out the hard way. See the resource file for the assembly format of the "DITL" items.

The Application Program

Our application itself is very simple. It is a static window display of a bunch of boxes created with the `QDSTUFF` subroutine, called whenever an update event for our window occurs. To make it a little more interesting, the last box drawn is also filled with a pattern. Patterns are simply 8 bytes of hex information repeated in the rectangle. A count variable keeps track of the number of boxes drawn and a loop calls `_FrameRect` until all the boxes are displayed. Don't forget to re-set the count for the next update event! Figure 3 shows the application program output and our OPEN/CLOSE flip-flop menu items. These serve to hide or display our application window in a fashion similar to the go away box. Don't forget, if the window goes away, you have to have some method of opening it again! Hence the OPEN and CLOSE file menu items.

Finally, in figure 4, is our linker file, pretty much the same format as last month. Remember that the signature byte, "GLAD" appears both in the resource file's bundle and identification resources as well as in the linker file. Next month we use this same shell program, but we concentrate on the application, getting some dynamic animation in our application window. Stay tuned!

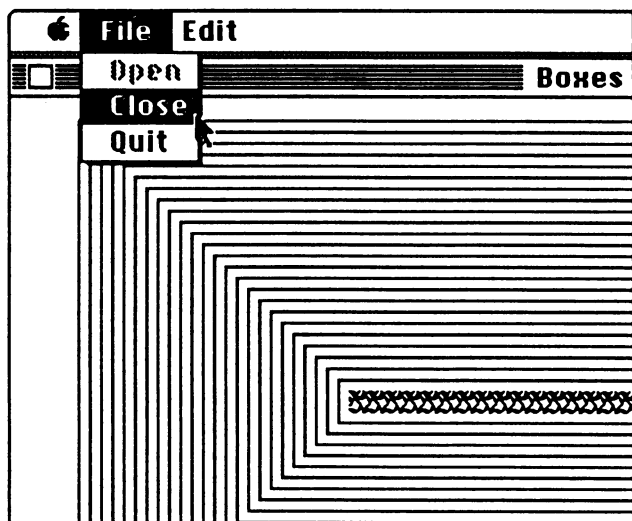


fig. 3 Program output

```
; EXAMPLE ASSEMBLY PROGRAM
```

```
; Boxes
```

```
; VERSION 23 JAN 85
```

```
; (C) MacTutor March 1985
```

```
; by David E. Smith
```

```
; Macro subset for Toolbox stuff
```

```
MACRO _InitGraf =      DC.W $A86E|
MACRO _InitWind =      DC.W $A912|
MACRO _NewWindow =     DC.W $A913|
MACRO _setport =       DC.W $A873|
MACRO _InitFont =      DC.W $A8FE|
MACRO _InitMenu =      DC.W $A930|
MACRO _InitDialog =    DC.W $A97B|
MACRO _TEInit =        DC.W $A9CC|
MACRO _Initpack =      DC.W $A9E5|
MACRO _FlushEvents =   DC.W $A032|
MACRO _InitCursor =    DC.W $A850|
MACRO _GetNextEvent =  DC.W $A970|
MACRO _FrameRect =     DC.W $A8A1|
MACRO _TextFont =      DC.W $A887|
MACRO _TextFace =      DC.W $A888|
MACRO _TextSize =      DC.W $A88A|
MACRO _MoveTo =        DC.W $A893|
```

```
MACRO _DrawString =    DC.W $A884|
MACRO _PenSize =       DC.W $A89B|
MACRO _EraseRect =     DC.W $A8A3|
MACRO _GetRMenu =      DC.W $A9BF|
MACRO _InsertMenu =    DC.W $A935|
MACRO _DrawMenuBar =   DC.W $A937|
MACRO _Debugger =      DC.W $A9FF|
MACRO _NewMenu =       DC.W $A931|
MACRO _AppendMenu =    DC.W $A933|
MACRO _AddResMenu =    DC.W $A94D|
MACRO _FindWindow =    DC.W $A92C|
MACRO _DragWindow =    DC.W $A925|
MACRO _SystemTask =    DC.W $A9B4|
```

```
MACRO _MenuSelect =    DC.W $A93D|
MACRO _HiLiteMenu =    DC.W $A938|
MACRO _EnableItem =    DC.W $A939|
MACRO _DisableItem =   DC.W $A93A|
MACRO _BeginUpdate =   DC.W $A922|
MACRO _EndUpdate =     DC.W $A923|
MACRO _SystemClick =   DC.W $A9B3|
MACRO _FrontWindow =   DC.W $A924|
MACRO _SelectWindow =  DC.W $A91F|
MACRO _GetItem =       DC.W $A946|
MACRO _OpenDeskAcc =   DC.W $A9B6|
MACRO _TrackGoAway =   DC.W $A91E|
MACRO _HideWindow =    DC.W $A916|
MACRO _PenNormal =     DC.W $A89E|
MACRO _ShowWindow =    DC.W $A915|
MACRO _TENew =         DC.W $A9D2|
MACRO _TECut =         DC.W $A9D6|
MACRO _TECopy =        DC.W $A9D5|
MACRO _TEPaste =       DC.W $A9DB|
MACRO _TEDelete =      DC.W $A9D7|
MACRO _TEActivate =    DC.W $A9D8|
MACRO _TEDeactivate =  DC.W $A9D9|
MACRO _SysEdit =       DC.W $A9C2|
MACRO _NewDialog =     DC.W $A97D|
MACRO _GetResource =   DC.W $A9A0|
MACRO _IsDialogEvent = DC.W $A97F|
MACRO _DialogSelect =  DC.W $A980|
MACRO _CloseWindow =   DC.W $A92D|
MACRO _FillRect =      DC.W $A8A5|
```

```
; DECLARE LABELS EXTERNAL
```

```
XDEF START ; required for linker
```

```
; LOCAL EQUATES
```

```
MouseDowneq 1
MaxEvents eq 12
AllEvents eq $0000FFFF
TEXTHANDLE eq A2
```

```
; MAIN PROGRAM SEGMENT
```

```
DC.W 'DAVE' ;find start of program
```

```
; --- SAVE THE WORLD -----
```

```
START: MOVEM.L D0-D7/A0-A6, -(SP)
        LEASAVEREQS,A0
        MOVE.L A6,(A0) ; local var
        MOVE.L A7,4(A0) ; stack ptr
```

```
; ---INITIALIZE ALL MANAGERS-----
```

```
; SET UP QUICKDRAW GLOBALS
```

```
PEA -4(A5) ;push qd global ptr
_InitGraf ;init quickdraw global
```

```
;--- SET UP REMAINING MANAGERS ---
```

```
_InitFont ; init font manager
```

```
_InitWind    ; init window manager
_InitMenu    ; init menu manager

CLR.L -(SP)    ; kill the restart
_InitDialog    ; init dialog manager
```

```
_TEInit        ; init text edit (ROM)
```

```
MOVE.W #2,-(SP) ; set-up
_Initpack        ; init package mgr
```

```
MOVE.L #AllEvents,D0 ; all events
_FlushEvents        ; flushed
```

```
_InitCursor    ; make cursor the arrow
```

SET UP MENU BAR WITH POP-UP MENUS

```
CLR.L -(SP)        ; returned menu handle
MOVE.W #1,-(SP)    ; push menu ID 1 Apple
_GetRMenu          ; get menu from resource
LEA APPLEMENU,A0    ; copy apple menu handle
MOVE.L (SP),(A0)    ; to local data block
MOVE.L (SP), -(SP)  ; push handle for desk AC
                    ; menu handle still on stack
CLR.W -(SP)        ; push 0 for append
_InsertMenu        ; add menu item for About...
```

```
MOVE.L #'DRVR', -(SP) ; load DRVR type res
_AddResMenu        ; add desk acc
                    ; from system file.
CLR.L -(SP)        ; return menu handle
MOVE.W #2,-(SP)    ; push menu ID 2 File
_GetRMenu          ; get menu from resource
```

```
LEA FILEMENU,A0    ; copy FILE menu handle
MOVE.L (SP),(A0)    ; to local data block
                    ; leave handle on stack
CLR.W -(SP)        ; append to menu
_InsertMenu        ; the File stuff
```

```
CLR.L -(SP)        ; return menu handle
MOVE.W #3,-(SP)    ; push menu ID 3 Edit
_GetRMenu          ; get menu from resource
                    ; leave handle on stack
LEA EDITMENU,A0    ; copy EDIT menu handle
MOVE.L (SP),(A0)    ; to local data block
```

```
CLR.W -(SP)        ; append to menu
_InsertMenu        ; the File stuff
```

```
_DrawMenuBar        ; our menu bar!
```

SET UP NEW HEAP WINDOW ----

```
CLR.L -(SP)        ; return window ptr
CLR.L -(SP)        ; window record ptr(HEAP)
PEA WBOUNDS        ; window rectangle size
PEA WINDTITLE      ; window title
MOVE.W #$100,-(SP) ; true = visible
```

```
MOVE.W #0,-(SP)    ; doc type window
MOVE.L #-1,-(SP)    ; window in front
MOVE.W #$100,-(SP)  ; true=closebox
MOVE.L #0,-(SP)    ; reference value
_NewWindow        ; make new window
```

ACTIVATE THIS NEW WINDOW ----

```
LEA WPOINTER,A0    ; copy our window ptr
MOVE.L (SP),(A0)    ; to stacksave
                    ; ptr still on stack.
_setport            ; set to current window
```

SET UP TEXT EDIT ABILITY ----

```
CLR.L -(SP)        ; text handle returned
PEA DestRect        ; destination for text
PEA ViewRect        ; viewing of text
_TENew              ; new text record on heap
MOVE.L (SP)+,TEXTHANDLE ; pop handle
```

INITIALIZE VARIABLES -----

```
JSR INITVAR        ; INITIALIZE DS STORAGE
```

EVENT LOOP -----

GetEvent:

```
_SystemTask        ; check out desk acc

CLR -(SP)          ; returned event
MOVE #AllEvents,-(SP) ; mask all events
PEA EventRecord     ; event record block
_GetNextEvent       ; go check the mouse
MOVE (SP)+,D0       ; get event result
CMP #0,D0           ; if 0 then no event
BEQ GetEvent        ; loop until it happens
```

check for dialog event

```
CLR -(SP)          ; result
PEA EventRecord     ; push event
_IsDialogevent      ; was it a dialog event?
MOVE (SP)+,D0       ; get result
CMP #0,D0           ; is it false?
BEQ Processevent    ; yes, not dialog
                    ; no, was dialog event
```

```
CLR -(SP)          ; result
PEA EventRecord     ; event
PEA ABOUTHANDLE     ; dialog handle
PEA ITEMHIT         ; item hit
_Dialogselect       ;
MOVE (SP)+,D0       ; get result
CMP #0,D0           ; is it false?
BEQ GetEvent        ; yes
```

```
; note: always false since no
; controls enabled. closebox is
; returned false.
```

```
_Debugger        ; do we ever get here?
```

JMP GetEvent

Processevent:

; JUMP TABLE OF EVENT PROCESSING

```
CLR.L D0
MOVE What,D0      ;what to do! (event #)
CMP  #MaxEvents,D0 ;event no. >11?
BGE  GetEvent      ;yes so ignore it
ADD  D0,D0         ;multiply by 2
MOVE EventTable(D0),D0
JMP  EventTable(D0) ;execute event
```

EventTable:

```
DC.W  GetEvent-EventTable ;null event #0
DC.W  ButtonDown-EventTable ;mouse down#1
DC.W  GetEvent-EventTable ;mouse up #2
DC.W  GetEvent-EventTable ;key down #3
DC.W  GetEvent-EventTable ;key up #4
DC.W  GetEvent-EventTable ;auto key #5
DC.W  Update-EventTable ;update event #6
DC.W  GetEvent-EventTable ;Disk Event #7
DC.W  Activate-EventTable ;activate #8
DC.W  GetEvent-EventTable ;Abort #9
DC.W  GetEvent-EventTable ;Network #10
DC.W  GetEvent-EventTable ;I/O Driver #11
DC.W  GetEvent-EventTable ;Appl. event 1
DC.W  GetEvent-EventTable ;Appl. event 2
DC.W  GetEvent-EventTable ;Appl. event 3
DC.W  GetEvent-EventTable ;Appl. event 4
```

; note that linker error may result if
; the table entry is >128

; --- EVENT PROCESSING ROUTINES ---

Activate:

; window needs attention

```
MOVE.L WPOINTER,A0 ;copy window ptr.
CMP.L Message,A0   ;Event our window?
BNE  GetEvent      ;no
```

```
MOVE.W Modify,D0   ;yes, check flags
```

```
BTST #0,D0         ;check activate bit
BEQ  Deactivate     ;unactivate window
```

; activate our window

```
MOVE.L TEXTHANDLE,-(SP)
_TEActivate          ;activate text edit

MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #1,-(SP)      ;push menu item 1
_DisableItem         ;disable undo
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #3,-(SP)      ;push menu item
_DisableItem         ;disable cut
```

```
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #4,-(SP)      ;push menu item
_DisableItem         ;disable copy
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #5,-(SP)      ;push menu item
_DisableItem         ;disable paste
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #6,-(SP)      ;push menu item
_DisableItem         ;disable clear
```

```
MOVE.L WPOINTER,-(SP) ;push window ptr
_SetPort              ;make our port active
JMP  GetEvent
```

Deactivate:

```
MOVE.L TEXTHANDLE,-(SP)
_TEDeactivate        ;turn off text edit
                    ;let desk acc. use it

MOVE.L EDITMENU,-(SP)
MOVE.W #1,-(SP)
_EnableItem          ;enable undo
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #3,-(SP)      ;push menu item
_EnableItem          ;able cut
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #4,-(SP)      ;push menu item
_EnableItem          ;enable copy
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #5,-(SP)      ;push menu item
_EnableItem          ;enable paste
MOVE.L EDITMENU,-(SP);push menu handle
MOVE.W #6,-(SP)      ;push menu item
_EnableItem          ;enable clear
JMP  GetEvent
```

Update:

; window needs refreshing

```
MOVE.L WPOINTER,-(SP) ;push window ptr
_BeginUpdate
MOVE.L WPOINTER,-(SP) ; our window ptr.
_SetPort              ;restore our port
                    ;least an update draw in
                    ;the wrong window.
BSR  QDSTUFF          ;re-draw everything
```

```
MOVE.L WPOINTER,-(SP)
_EndUpdate
JMP  GetEvent
```

ButtonDown:

```
CLR  -(SP)           ;result of findwindow
MOVE.L Point, -(SP)  ;push mouse coord.
PEA  WWindow         ;push holder window handle
_FindWindow          ;where was mouse click?
CLR.L D0
MOVE (SP)+,D0        ;pop region number
ADD  D0,D0           ;multiply by 2
MOVE WindowTable(D0),D0
```

JMP WindowTable(D0) ;do mouse down event

WindowTable: ;table of button down events

DC.W GetEvent-WindowTable ;in desktop
DC.W Menubar-WindowTable ;in menu bar
DC.W System-WindowTable ;in system window
DC.W Content-WindowTable ;in content region
DC.W Drag-WindowTable ;in drag region
DC.W GetEvent-WindowTable ;in grow region (ignored)
DC.W Hide-WindowTable ;in go-away region

System:

; Button pressed in a system window
; _SystemClick calls the desk acc.

PEA EventRecord ;push event record
MOVE.L WWindow,-(SP) ;push find window
_SystemClick ;let system do desk
JMP GetEvent

Content:

MOVE.L WWindow,-(SP) ;activate window
_SelectWindow ;make it front
JMP GetEvent

Drag: ;mouse button in drag
;could be dialog window

MOVE.L WWindow,-(SP) ;push find window pt
MOVE.L Point,-(SP) ;push mouse coord
PEA DRAGWINDOW ;window boundries
_DragWindow ;move window

MOVE.L WWindow,A0 ;find window ptr.
CMP.L ABOUTHANDLE,A0 ;is it about window?
BNE ENDDRAG ;no, exit
JSR INITVAR ;yes, reset about
;window location

ENDDRAG:

JMP GetEvent ;return to event loop

Hide: ;close box
;could be dialog window

CLR.W -(SP) ;boolean result
MOVE.L WWindow,-(SP) ;push find window
MOVE.L Point,-(SP) ;push mouse loc
_TrackGoAway
MOVE.W (SP)+,D0 ;get result
BEQ GetEvent ;false result

MOVE.L WWindow,A0 ;find window ptr.
CMP.L WPOINTER,A0 ;is it our window?
BEQ Hideours ;yes, hide ours

MOVE.L WWindow,-(SP) ;must be about
_CloseWindow ;close it

;note: Don't use closedialog for modeless
;dialog windows if more than one present.

; update about window location

LEA DBOUNDS,A0 ;set-up our globals
LEA Savetop,A1

MOVE.W 0(A0),D0 ;TOP VALUE
MOVE.W 0(A1),D1 ;original value
CMP.W D1,D0 ;past init value?
BLE GetEvent ;skip update

ADDI #-20,D0 ;decrement top
MOVE.W D0,(A0)

MOVE.W 2(A0),D0 ;LEFT VALUE
ADDI #-20,D0
MOVE.W D0,2(A0)

MOVE.W 4(A0),D0 ;BOTTOM VALUE
ADDI #-20,D0
MOVE.W D0,4(A0)

MOVE.W 6(A0),D0 ;RIGHT VALUE
ADDI #-20,D0
MOVE.W D0,6(A0)

JMP GetEvent

Hideours:

MOVE.L WPOINTER,-(SP) ;our window ptr
_HideWindow ;hide window
MOVE.L FILEMENU,-(SP) ;menu handle
MOVE.W #1,-(SP) ;open item
_EnableItem ;enable open for file menu
MOVE.L FILEMENU,-(SP) ;push menu handle
MOVE.W #2,-(SP) ;close
_DisableItem ;disable it
JMP GetEvent

Menubar:

CLR.L -(SP) ;returned menu choice
MOVE.L Point,-(SP) ;mouse position
_MenuSelect ;Find menu selected
MOVE (SP)+,D5 ;pop menu ID to D5
MOVE (SP)+,D6 ;pop menu item ID
CLR -(SP) ;select all menus
_HiLiteMenu ;unhilite them all

CMP #1, D5 ;apple menu?
BEQ Inapplemenu
CMP #2, D5 ;file menu?
BEQ Infilemenu
CMP #3, D5 ;edit menu?
BEQ Ineditmenu
BRA GetEvent ;no place else

Inapplemenu:

CMP #1, D6 ;is it about...?
BEQ About

; must be a desk accessory

```
MOVE.L  APPLEMENU, -(SP) ;push apple menu
MOVE.W  D6, -(SP)        ;push item number
PEA  DeskName      ;push holder for desk name
_GetItem                      ;find desk acc. name
```

```
CLR  -(SP)                ;result
PEA  DeskName             ;push desk acc. name
_OpenDeskAcc              ;do it
MOVE (SP)+, D0            ;pop result
MOVE.L  WPOINTER, -(SP)  ;push window ptr.
_SetPort                  ;restore our port
JMP  GetEvent
```

About:

```
CLR.L  -(SP)              ;return handle
MOVE.L  #'DITL', -(SP)    ;item list
MOVE.W  #256, -(SP)       ;ID
_GetResource      ;load item list resource
LEA  DIALOGITEMS, A0      ;handle storage
MOVE.L  (SP)+, (A0)       ;save handle
```

; update window location

```
LEA  DBOUNDS, A0          ;set-up our globals
```

```
MOVE.W  0(A0), D0         ;TOP VALUE
ADDI  #20, D0
MOVE.W  D0, (A0)
```

```
MOVE.W  2(A0), D0         ;LEFT VALUE
ADDI  #20, D0
MOVE.W  D0, 2(A0)
```

```
MOVE.W  4(A0), D0         ;BOTTOM VALUE
ADDI  #20, D0
MOVE.W  D0, 4(A0)
```

```
MOVE.W  6(A0), D0         ;RIGHT VALUE
ADDI  #20, D0
MOVE.W  D0, 6(A0)
```

;SET UP MODELESS DIALOG ON HEAP

```
CLR.L  -(SP)              ;return window ptr
CLR.L  -(SP)              ;window record ptr. (HEAP)
PEA  DBOUNDS              ;window rectangle size
PEA  DWINDTITLE           ; window title
MOVE.W  #$100, -(SP)      ; true = visible
MOVE.W  #16, -(SP)        ; round doc type window
MOVE.L  #-1, -(SP)        ; window in front
MOVE.W  #$100, -(SP)      ; true=closebox
MOVE.L  #0, -(SP)         ; reference value
MOVE.L  DIALOGITEMS, -(SP) ;items list handle
_NewDialog                ; make new window
```

```
LEA  ABOUTHANDLE, A0      ;dialog window handle
MOVE.L  (SP), (A0)        ; save handle
_SetPort                  ;make dialog active
```

```
MOVE.L  TEXTRHANDLE, -(SP) ;deactivate text
_TEDeactivate
JMP  GetEvent
```

Infilemenu:

```
CMP  #1, D6              ;Open?
BEQ  Open
CMP  #2, D6              ;Close?
BEQ  Close
CMP  #3, D6              ;quit?
BNE  GetEvent            ;no, try again
BRA  Exit                ;yes, exit
```

Open:

; window is hiding...open it again

```
MOVE.L  WPOINTER, -(SP) ;our window ptr
_ShowWindow
MOVE.L  FILEMENU, -(SP) ;menu handle
MOVE.W  #2, -(SP)        ;open item
_EnableItem              ;enable open for file menu
MOVE.L  FILEMENU, -(SP) ;push handle
MOVE.W  #1, -(SP)        ;open
_DisableItem             ;disable it
JMP  GetEvent
```

Close:

; perform go away function

```
MOVE.L  WPOINTER, -(SP) ;our window ptr
_HideWindow              ;hide window
MOVE.L  FILEMENU, -(SP) ;menu handle
MOVE.W  #1, -(SP)        ;open item
_EnableItem              ;enable open
MOVE.L  FILEMENU, -(SP) ;push handle
MOVE.W  #2, -(SP)        ;close
_DisableItem             ;disable it
JMP  GetEvent
```

Ineditmenu:

; check if system edit in desk acc.

```
CLR  -(SP)               ;result
MOVE.W  D6, -(SP)        ;push menu item
SUBQ  #1, (SP)            ;system edit offset
_SysEdit                  ;do editing
MOVE.B  (SP)+, D0         ;pop result
BNE  GetEvent            ;system edit did it
                        ;desk acc. not active
```

```
CLR.L  -(SP)
_FrontWindow              ;whoose up?
MOVE.L  (SP)+, A0
CMP.L  ABOUTHANDLE, A0    ;is this an about ?
BEQ  GetEvent            ;yes, ignore it
```

```
CMP  #3, D6              ;our cut?
```

```
BEQ  CUT
CMP  #4,D6          ;our copy?
BEQ  COPY
CMP  #5,D6          ;our paste?
BEQ  PASTE
CMP  #6,D6          ;our clear?
BEQ  CLEAR
```

```
JMP  GetEvent      ;nothing left
```

; - text editing only for desk acc. -

```
CUT:
MOVE.L  TEXTHANDLE,-(SP) ;text handle
_TECut
JMP  GetEvent
```

```
COPY:
MOVE.L  TEXTHANDLE,-(SP) ;text handle
_TECopy
JMP  GetEvent
```

```
PASTE:
MOVE.L  TEXTHANDLE,-(SP) ;text handle
_TEPaste
JMP  GetEvent
```

```
CLEAR:
MOVE.L  TEXTHANDLE,-(SP) ;text handle
_TEDelete
JMP  GetEvent
```

Exit:

```
JMP  Tofinder      ;return to finder
```

; ----- END OF MAIN -----

; ----- INITVAR SUBROUTINE -----

INITVAR:

```
LEA  DBOUNDS(A5),A0 ;set-up dialog windo
                        ;box frame size.
MOVE.W #70, (A0)      ;TOP
MOVE.W #70, 2(A0)     ;LEFT
MOVE.W #220, 4(A0)    ;BOTTOM
MOVE.W #370, 6(A0)    ;RIGHT
```

```
LEA  Savetop(A5),A1
MOVE.W (A0), (A1)      ;GET TOP
```

```
LEA  COUNT(A5),A1
MOVE.W #24, (A1)
RTS
```

; ---- QDSTUFF SUBROUTINE -----

QDSTUFF:

```
; This subroutine is a static
; window display of boxes
```

_PenNormal

```
LEA  top(A5),A0      ;set-up appl. window
                        ;box frame size.
MOVE.W #10, (A0)      ;TOP
MOVE.W #30, 2(A0)    ;LEFT
MOVE.W #260, 4(A0)   ;BOTTOM
MOVE.W #450, 6(A0)   ;RIGHT
```

```
PEA  top(A5)         ;frame rectangle
_FrameRect          ;draw rectangle
```

BOXLOOP:

```
LEA  top(A5),A0      ; do a bunch boxes
LEA  COUNT(A5),A4
ADDQ #5,(A0)
ADDQ #5,2(A0)
SUBQ #5,4(A0)
SUBQ #5,6(A0)
PEA  top(A5)
_FrameRect
SUBQ #1,(A4)
BNE  BOXLOOP        ;loop til all drawn
```

```
PEA  top(A5)
PEA  boxpattern
_FillRect            ;fill last box
```

```
LEA  COUNT(A5),A1
MOVE.W #24, (A1)      ;reset count!
RTS
```

; --- RESTORE THE WORLD -----

```
Tofinder: LEA  SAVEREGS,A0 ; get 'em back
MOVE.L (A0),A6 ; local var
MOVE.L 4(A0),A7 ;restore stack
MOVEM.L (SP)+,D0-D7/A0-A6
```

; ---- RETURN TO FINDER -----

```
RTS ; return to finder
```

; ----LOCAL DATA AREA -----

```
APPLEMENU: DC.L 0 ;apple menu handle
FILEMENU:  DC.L 0 ;file menu handle
EDITMENU:  DC.L 0 ;edit menu handle
DeskName:  DCB.W 16,0 ;Desk accesorys name
SAVEREGS:  DCB.L 2,0 ;set save area
boxpattern: DC.W $1234,$ABCD,$DCBA,$4321
```

; ----- our window definitions -----

```
WPOINTER:  DC.L 0 ;store window pt

WBOUNDS:DC.W 40 ;TOP -windo rect
DC.W 2 ;LEFT
DC.W 335 ;BOTTOM
DC.W 508 ;RIGHT
```

```
ViewRect:  DC.W 50 ;text edit view rect
DC.W 4
```

```

        DC.W 245
        DC.W 405

DestRect:    DC.W 50      ;text edit dest rect
             DC.W 4
             DC.W 245
             DC.W 405

WINDTITLE:   DC.B 5       ; title length
             DC.B 'Boxes',0

DRAGWINDOW:  DC.W 30      ;rectangle
             DC.W 1       ;for dragging
             DC.W 350
             DC.W 510

; ---- our dialog window definitions ----

ABOUTHANDLE: DC.L 0      ;handle holder

DWINDTITLE:   DC.B 30    ; title length
DC.B 'All about the Boxes program...',0

ITEMHIT:      DC.W 0      ;ITEM HIT HOLDER

DIALOGITEMS:  DC.L 0      ;handle holder for
              ; resource DITL
; -----

EventRecord:

    What:      DC.W 0      ;what event number
    Message:   DC.L 0      ;ptr. to msg
    When:      DC.L 0      ;Time event was posted
    Point:     DC.L 0      ;mouse coordinates
    Modify:    DC.W 0      ;state of keys & button

WWindow:      DC.L 0      ;Find windos result

; ---- APPLICATION GLOBALS ----

top:          DS.W 1       ;frame size storage
left:         DS.W 1       ;for QDSTUFF rect
bottom:       DS.W 1
right:        DS.W 1

DBOUNDS: DS.W 1           ;TOP-windo size rect
         DS.W 1           ;LEFT
         DS.W 1           ;BOTTOM
         DS.W 1           ;RIGHT

Savetop:     DS.W 1       ;save top value

COUNT:      DS.W 1       ;BOX COUNT

; --- END OF PROGRAM -----

; Boxes_rscs.asm
; resource file for the Boxes

```

```

; (C) MacTutor March 1985
; created using the assembler
; signature is creator tag
;
RESOURCE 'GLAD' 0 'IDENTIFICATION'

        DC.B 22, 'BOXES -- JAN. 23 1984 '

.ALIGN 2
RESOURCE 'BNDL' 128 'BUNDLE'

        DC.L 'GLAD'      ;NAME OF SIGNATURE
        DC.W 0,1         ;DATA (DOESN'T CHANGE)
        DC.L 'ICN#'      ;ICON MAPPINGS
        DC.W 0           ;NUMBER OF MAPPINGS-1
        DC.W 0,128       ;MAP 0 TO ICON 128

        DC.L 'FREF'      ;FREF MAPPINGS
        DC.W 0           ;NUMBER OF MAPPINGS-1
        DC.W 0,128       ;MAP 0 TO FREF 128

RESOURCE 'FREF' 128 'FREF 1'

        DC.B 'APPL', 0, 0, 0

.ALIGN 2
RESOURCE 'ICN#' 128 'MY ICON'

; FIRST APPLICATION ICON BIT MAP

INCLUDE Boxes.icon.asm

; MENU BAR RESOURCES

.ALIGN 2
RESOURCE 'MENU' 1 'apple menu'

        DC.W 1           ;MENU ID
        DC.W 0           ;WIDTH HOLDER
        DC.W 0           ;HEIGHT HOLDER
        DC.L 0           ;STD. MENU PROC HOLDER
        DC.L $1FF        ;ENABLE ALL ITEMS
        DC.B 1           ;TITLE LENGTH
        DC.B 20          ;APPLE SYMBOL

        DC.B 21          ;MENU ITEM LENGTH
        DC.B 'ABOUT THIS PROGRAM...'
        DC.B 0           ;NO ICON
        DC.B 0           ;NO KEYBOARD EQUIVALENT
        DC.B 0           ;MARKING CHARACTER
        DC.B 0           ;STYLE OF ITEM'S TEXT

        DC.B 0           ;END OF MENU ITEMS

.ALIGN 2
RESOURCE 'MENU' 2 'file menu'

        DC.W 2           ;MENU ID
        DC.W 0           ;WIDTH HOLDER
        DC.W 0           ;HEIGHT HOLDER
        DC.L 0           ;RESOURCE HOLDER
        DC.L $D          ;ENABLE ALL ITEMS except 1

```

```

DC.B 4      ; TITLE LENGTH
DC.B 'File'  ; file menu

DC.B 4      ;MENU ITEM LENGTH
DC.B 'Open'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

DC.B 5      ;MENU ITEM LENGTH
DC.B 'Close'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

DC.B 4      ;MENU ITEM LENGTH
DC.B 'Quit'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

DC.B 0      ; END OF MENU ITEMS

```

```

.ALIGN 2
RESOURCE 'MENU' 3 'edit menu'

```

```

DC.W 3      ;MENU ID
DC.W 0      ;WIDTH HOLDER
DC.W 0      ;HEIGHT HOLDER
DC.L 0      ;RESOURCE ID OF PROC
DC.L $3B    ;ENABLE ALL ITEMS except 2 & 6
DC.B 4      ; TITLE LENGTH
DC.B 'Edit'  ; edit menu

DC.B 4      ;MENU ITEM LENGTH
DC.B 'Undo'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

```

```

DC.B 9      ;MENU ITEM LENGTH
DC.B '-----'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

```

```

DC.B 3      ;MENU ITEM LENGTH
DC.B 'Cut'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

```

```

DC.B 4      ;MENU ITEM LENGTH
DC.B 'Copy'
DC.B 0      ; NO ICON

```

```

DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

```

```

DC.B 5      ;MENU ITEM LENGTH
DC.B 'Paste'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

```

```

DC.B 5      ;MENU ITEM LENGTH
DC.B 'Clear'
DC.B 0      ; NO ICON
DC.B 0      ; NO KEYBOARD EQUIVALENT
DC.B 0      ; NO MARKING CHARACTER
DC.B 0      ; STYLE OF ITEM'S TEXT

```

```

DC.B 0      ; END OF MENU ITEMS

```

```

.ALIGN 2
RESOURCE 'ICON' 256 'DIALOG ICON'

```

```

; Dialog box icon bit map

```

```

INCLUDE Boxes.icon.asm

```

```

; ---- DIALOG ITEMS LIST -----
;
; text in dialog items must be of even
; length. Item types are as follows:
;
; control item = control item + 4
; button control = 0
; check control = 1
; radio button control = 2
; resource control = 3
;
; static text = 8
; edit text = 16
; icon item = 32
; quickdraw pict item = 64
; user item = 0
;
; disable item = item type + 128
;
; resource ID's for system icons
;
; stop icon = 0
; note icon = 1
; caution icon = 2

```

```

.ALIGN 2
RESOURCE 'DITL' 256 'Boxes DIALOG'

```

```

DC.W 4      ;number of items -1

```

```

DC.L 0      ;handle holder item 1
DC.W 10     ;rectangle-top
DC.W 20     ;left
DC.W 25     ;bottom
DC.W 375    ;right

```

```
DC.B 128+8;TYPE-STATIC TEXT + disable
DC.B 26 ;TEXT LENGTH
DC.B 'This program is a complete'
```

```
DC.L 0 ;handle holder item 2
DC.W 30 ;rectangle-top
DC.W 20 ;left
DC.W 45 ;bottom
DC.W 375 ;right
DC.B 128+8;TYPE-STATIC TEXT + disable
DC.B 30 ;TEXT LENGTH
DC.B 'application shell in assembly.'
```

```
DC.L 0 ;handle holder item 3
DC.W 50 ;rectangle-top
DC.W 200 ;left
DC.W 82;bottom
DC.W 232 ;right
DC.B 128+32;TYPE- icon + disable
DC.B 2 ;length of next data
DC.W 256 ;icon resource ID
```

```
DC.L 0 ;handle holder item 4
DC.W 50 ;rectangle-top
DC.W 20 ;left
DC.W 90 ;bottom
DC.W 60 ;right
DC.B 128+32;TYPE- icon + disable
DC.B 2 ;length of next data
DC.W 1 ;icon resource ID
```

```
DC.L 0 ;handle holder item 5
DC.W 110 ;rectangle-top
DC.W 20 ;left
DC.W 125 ;bottom
DC.W 375 ;right
DC.B 128+8;TYPE-STATIC TEXT + disable
DC.B 32 ;TEXT LENGTH
DC.B 'by David E. Smith 23 JAN 1985'
```

```
!START
[
)
/OUTPUT Boxes

Boxes

/TYPE 'APPL' 'GLAD'
/BUNDLE
/RESOURCES
Boxes_rscs

$
```

Fig. 4 Linker File



Assembly Language Lab

Animation Example



Animate Example

David E. Smith
Editor & Publisher
MacTutor Vol. 1 No. 5

Last month we presented a complete application shell program in assembly that supported desk accessories, cut and past and "about" dialog boxes. But the application itself was rather boring, simply drawing nested boxes in the window. This month, we concentrate on the application itself. To eliminate confusion, we will use the simple shell program from the December issue of MacTutor that simply opens a window. This allows us to concentrate on the application, which is to animate a paddle and ball. Once we understand animation, we can combine our paddle and ball program with the shell program published last month to create a complete Mac game application. But first, the problem of animation.

Animation Contest

The Mousehole recently held a programming contest sponsored by MacTutor on the problem of moving a paddle with the mouse. "Brett" offered an interesting solution: "Have the user blink a lot and pretend!" Another interesting solution offered by "Lone Falcon" and "Chief Wizard" and expanded on by "The Jerk" was to make the cursor a paddle and constrain the cursor to a given rectangle. The trap call "_PinRect" can be used to change the rectangle in which the cursor is allowed to move. Or the rectangle can be poked into the global variable at \$834 as mentioned by "The Jerk". One problem with this solution is that of providing an "escape" for the cursor when you want to go to the menu bar to quit the game!

A simple technique to move the paddle is to draw the paddle rectangle where the mouse now is and erase it from where the paddle used to be. The following example code shows how this might be done:

MOVEPADDLE:

```
; Copy current paddle to old paddle
LEA    paddle, A0
LEA    oldpaddle, A1
MOVE.W 0(A0), 0(A1)      ;top of rect
MOVE.W 2(A0), 2(A1)      ;left of rect
MOVE.W 4(A0), 4(A1)      ;bottom
MOVE.W 6(A0), 6(A1)      ;right
; Update paddle position from mouse
; _GetMouse trap previously called
; and mouse coord. saved in Mouse.
; Since paddle moves vertically, only
; the top and bottom coordinates need
; be updated.
LEA    Mouse, A0          ;get mouse
LEA    paddle, A1          ;get paddle
MOVE.W (A0), (A1)         ;update top
MOVE.W (A0), D0           ;get top
ADD.W  Paddlelength, D0    ;add length
```

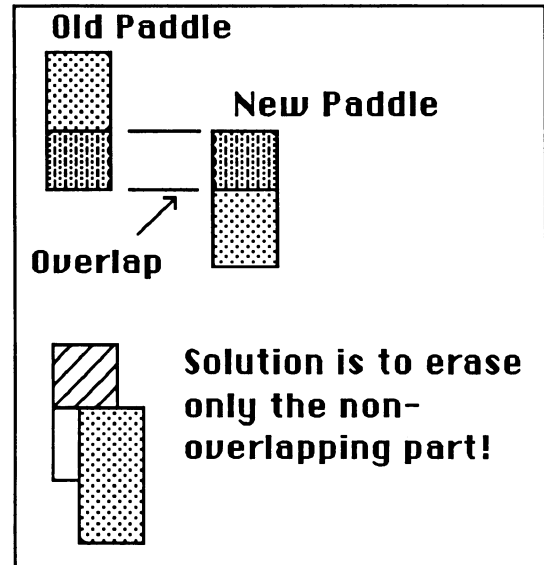


fig. 1

```
MOVE.W    D0, 4(A1)                ;update bottom
; Erase old paddle position
PEA        oldpaddle                ;push old pad.
_EraseRect ;erase it
; Draw new paddle
PEA        paddle                   ;push paddle
PEA        Paddlepattern            ;push fill pat.
_FillRect
RTS
```

The Paddle Blinks!

The problem with the above solution, and hence the contest, is that the paddle will blink as it moves up and down. The problem is to improve on the above technique to eliminate the blinking. The reason for the blinking is shown in figure 1.

As shown in figure 1, the paddle blinks because we are erasing and drawing part of the paddle that overlaps, causing it to blink on and off. The solution is to determine the non-overlapping part and erase only that part. One way to do this is to replace the bottom of the old paddle with the top of the new paddle. Then the modified old paddle rectangle would be only the cross-hatched area shown in figure 1 as the non-overlapping part. But that only works for the paddle moving down. If the paddle moves up, then the opposite condition applies as shown in the following code segment:

ERASEPADDLE:

```
; create difference rectangle
LEA    paddle, A0                ;new paddle
LEA    oldpaddle, A1            ;old paddle
```

```

MOVE.W (A0), D0 ;get new paddle
MOVE.W (A1), D1 ;get old paddle
SUB.W D1, D0 ;top difference
BMI UP ;moving up?
DN: MOVE.W (A0), 4(A1) ;update bottom
    JMP Eraseit
UP: MOVE.W 4(A0), (A1) ;update top
Eraseit: PEA oldpaddle
        _EraseRect

```

Contest Winner!

The above method works fine for a rectangular paddle moving up or down. But what about the ball? The winning solution to our animation contest was posted by "Don L." in which he explained how regions could be used in a manner similar to what was done above for rectangles. Moreover, the toolbox has a special trap for finding the difference of two regions but does not have a similar trap for rectangles. The solution then, is to form a union region of the old and new paddle regions, and then take the difference between the new paddle and this union region. The difference region is then erased and the paddle re-drawn. The result is a non-blinking paddle but using regions instead of rectangles.

In our example program we use "Don L."s method to move the ball as a region, and the rectangular method for moving the less complicated paddle rectangle. See the subroutine MOVEPADDLE for the rectangular implementation discussed above and the subroutine MOVEBALL for the region solution submitted by Don. The tricky part is getting the regions properly defined with NewRgn, OpenRgn and CloseRgn.

Detecting Collisions

Once we have a moving paddle responding to the mouse, and a ball moving towards us, we need to detect if the ball has hit the paddle. A simple way to do this is to use the trap _RectInRgn, which tells if the given rectangle intersects with the given region. We have used this trap by pushing the paddle rectangle as the given rectangle, and the ball region as the given region and tested the returning byte for true or false. If true, then the ball has "hit" the paddle, and we erase the ball and start a new one. If not, then the paddle has "missed" the ball and the game ends on a click of the mouse. Of course, in the final game, we really want the ball to "bounce" and move away from the paddle if there is a hit. We will leave the problem of bouncing for next month.

Animate Example

Our sample program includes all of the topics we've discussed this month in a complete program. To conserve space, none of the fancy user interface support we did last month is included. The program opens a window and displays our game field with a rectangular paddle, and launches a ball. If the user hits the ball, it disappears and another ball is launched. If the user misses the ball, the game stops until the button is pressed, at which point the finder is invoked. This is the basic animation required for a pinball type of game. Make note of the INITBALL routine that sets up the three temporary regions for use with the ball, and the MOVEBALL routine

which erases the ball by only erasing the non-over-lapping region. Using these techniques, the paddle and ball should move smoothly over the shaded playing field.

Ball Speed

The speed of the ball is determined by the increment used to update the ball region. The trap "_OffSetRgn" is used to incrementally move the ball a small distance (dh, dv). In our example, dh is set at -1 and dv is zero, making the ball move horizontally one pixel left at a time. For a faster moving ball, the horizontal increment should be doubled in a loop, until the user misses. Try this and see how fast the ball appears to move. Note that moving the paddle very quickly does produce an effect on the speed of the ball slightly.

In figure 2 we see the output from this month's program. The white ball moves on a field of gray toward the black paddle. The white area was added in MacPaint to reduce the amount of gray, which does not print well.

Resources and Linker Files

The resource and linker files included are the minimum necessary to get an icon to pop up on the desktop. Our icon is shown at the top of the column header, and was produced using the Icon Converter Utility from issue 2 and "included" into our resource source code.

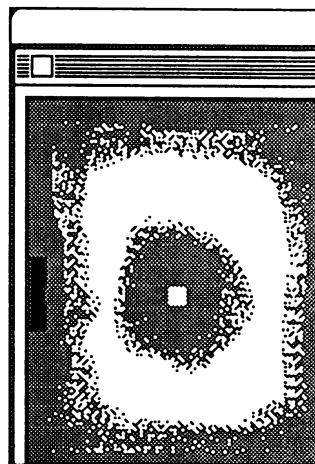


Fig. 2 Output

```

!START
[
]
/OUTPUT Animate Example

Animate

/TYPE 'APPL' 'ANIM'
/BUNDLE
/RESOURCES
Animate_rscs

$

```

Link File

```

; EXAMPLE ASSEMBLY PROGRAM
; ANIMATE (MacTutor 1-5)
; VERSION 27 FEB 1985
; (C) 1985 MacTutor by David E. Smith

```

; Macro subset for Toolbox stuff

```

MACRO _InitGraf = DC.W $A86E|
MACRO _InitWind = DC.W $A912|
MACRO _NewWindow = DC.W $A913|
MACRO _setport = DC.W $A873|
MACRO _InitFont = DC.W $A8FE|
MACRO _InitMenu = DC.W $A930|
MACRO _InitDialog = DC.W $A97B|
MACRO _TEInit = DC.W $A9CC|
MACRO _Initpack = DC.W $A9E5|

```

```

MACRO _FlushEvents = DC.W $A032|
MACRO _InitCursor = DC.W $A850|
MACRO _GetNextEvent = DC.W $A970|
MACRO _FrameRect = DC.W $A8A1|
MACRO _BackPat = DC.W $A87C|
MACRO _EraseRect = DC.W $A8A3|
MACRO _NewRgn = DC.W $A8D8|
MACRO _OpenRgn = DC.W $A8DA|
MACRO _CloseRgn = DC.W $A8DB|
MACRO _FrameRgn = DC.W $A8D2|
MACRO _FrameRoundRect = DC.W $A8B0|
MACRO _FillRect = DC.W $A8A5|
MACRO _FillRgn = DC.W $A8D6|
MACRO _BeginUpdate = DC.W $A922|
MACRO _EndUpdate = DC.W $A923|
MACRO _ClipRect = DC.W $A87B|
MACRO _GetMouse = DC.W $A972|
MACRO _CopyRgn = DC.W $A8DC|
MACRO _OffsetRgn = DC.W $A8E0|
MACRO _OffsetRect = DC.W $A8A8|
MACRO _UnionRgn = DC.W $A8E5|
MACRO _DiffRgn = DC.W $A8E6|
MACRO _MoveTo = DC.W $A893|
MACRO _DrawChar = DC.W $A883|
MACRO _RectInRgn = DC.W $A8E9|

```

; DECLARE LABELS EXTERNAL

```
XDEF START ; required for linker
```

; LOCAL EQUATES

```

MouseDown equ 1
AllEvents equ $0000FFFF
UpdateEvt equ 6

```

; SET UP MANAGERS

START:

```

PEA -4(A5) ;push qd global ptr
_InitGraf ;init quickdraw global
_InitFont ;init font manager
_InitWind ;init window manager
_InitMenu ;init menu manager
CLR.L -(SP) ;kill the restart
_InitDialog ;init dialog manager
_TEInit ;init text edit (ROM)
MOVE.W #2,-(SP) ;set-up
_Initpack ;init package mgr
MOVE.L #AllEvents,D0 ;all events
_FlushEvents ;flushed
_InitCursor ;make cursor the arrow

```

-- SET UP NEW WINDOW ON HEAP ---

```

CLR.L -(SP) ;return window ptr
CLR.L -(SP) ;window record ptr.
PEA WBOUNDS ;window rectangle
PEA WINDTITLE ;window title
MOVE.W #$100,-(SP) ;true = visible
MOVE.W #0,-(SP) ;doc type window

```

```

MOVE.L #-1,-(SP) ;window in front
MOVE.W #$100,-(SP) ;true=closebox
MOVE.L #0,-(SP) ;reference value
_NewWindow ;make new window

```

; -- ACTIVATE THIS NEW WINDOW -----

```

LEA WPOINTER,A0 ;copy window ptr
MOVE.L (SP),(A0) ;to stacksave
_setport ;current window

```

; INIT ALL VARIABLES

```
JSR INITVAR
```

; --EVENT LOOP -----

GetEvent:

```
JSR DOGAME ;move paddle with mouse
```

```

CLR -(SP) ;returned event
MOVE #AllEvents,-(SP) ;mask all events
PEA EventRecord ;event record block
_GetNextEvent ;go check the mouse
MOVE (SP)+,D0 ;get event result
CMP #0,D0 ;if 0 then no event
BEQ GetEvent ;loop until it happens

```

; JUMP TABLE OF EVENT PROCESSING

```

MOVE What,D0 ;what to do!
CMP #MouseDown,D0 ;button down?
BEQ EXIT ;yes so exit...
CMP #UpdateEvt,D0 ;is it an update event?
BEQ UPDATE ;yes, so do it
JMP GetEvent ;get next event

```

UPDATE:

; window needs refreshing

```

MOVE.L WPOINTER,-(SP) ;push window ptr
_BeginUpdate
MOVE.L WPOINTER,-(SP) ;push our window ptr.
_SetPort ;restore our port
;least an update draw in
;the wrong window.
BSR QDSTUFF ;re-draw everything
MOVE.L WPOINTER,-(SP)
_EndUpdate
JMP GetEvent

```

; ----- END OF MAIN -----

INITVAR:

; INIT GAME VARIABLES

```
LEA Field.of.play(A5),A0 ;set-up appl. window
```

```

MOVE.W #5, (A0) ;Field.of.play TOP
MOVE.W #5, 2(A0) ;LEFT

```



```

MOVE.W #245, 4(A0) ;BOTTOM
MOVE.W #500, 6(A0) ;RIGHT

PEA Field.of.play(A5)
_ClipRect ;set clip region

LEA Paddle(A5),A0 ;set-up paddle
MOVE.W #6, (A0) ;TOP
MOVE.W #8, 2(A0) ;LEFT
MOVE.W #40, 4(A0) ;BOTTOM
MOVE.W #15, 6(A0) ;RIGHT

LEA Paddlelength(A5),A1 ;update paddle bottom
MOVE.W (A0),D0
MOVE.W 4(A0),D1
SUB.W D0,D1
MOVE.W D1, (A1)

; set up regions for ball

CLR.L -(SP)
_NewRgn ;set new region
LEA Ballrgn(A5), A1 ;make ball a region
MOVE.L (SP)+,(A1) ;save handle in ballrgn

CLR.L -(SP)
_NewRgn ;set new region
LEA oldballrgn(A5), A1 ;make old ball a region
MOVE.L (SP)+,(A1) ;save handle in oldballrg

CLR.L -(SP)
_NewRgn ;set new region
LEA unionrgn(A5), A1 ;make union ball a region
MOVE.L (SP)+,(A1) ;save handle in unionrgn

JSR INITBALL ;set-up ball

RTS

; ---- Init ball subroutine ----

INITBALL:

LEA Ball(A5),A0 ;set-up ball rect
MOVE.W #100, (A0) ;TOP
MOVE.W #300, 2(A0) ;LEFT
MOVE.W #109, 4(A0) ;BOTTOM
MOVE.W #309, 6(A0) ;RIGHT

LEA Ballwide(A5),A0 ;set up rounded corners
MOVE.W #6, (A0)
LEA Ballhigh(A5), A0
MOVE.W #6, (A0)

_OpenRgn ;init ball region
PEA Ball(A5)
MOVE.W Ballwide(A5),-(SP)
MOVE.W Ballhigh(A5),-(SP)
_FrameRoundRect
MOVE.L Ballrgn(A5),A0
MOVE.L A0,-(SP)
_CloseRgn

_OpenRgn ;init union region
PEA Ball(A5)
MOVE.W Ballwide(A5),-(SP)
MOVE.W Ballhigh(A5),-(SP)
_FrameRoundRect
MOVE.L unionrgn(A5),A0
MOVE.L A0,-(SP)
_CloseRgn
RTS

```

```

_CloseRgn

_OpenRgn ;init oldball region
PEA Ball(A5)
MOVE.W Ballwide(A5),-(SP)
MOVE.W Ballhigh(A5),-(SP)
_FrameRoundRect
MOVE.L oldballrgn(A5),A0
MOVE.L A0,-(SP)
_CloseRgn

_OpenRgn ;init union region
PEA Ball(A5)
MOVE.W Ballwide(A5),-(SP)
MOVE.W Ballhigh(A5),-(SP)
_FrameRoundRect
MOVE.L unionrgn(A5),A0
MOVE.L A0,-(SP)
_CloseRgn
RTS

DOGAME:

; update game field during null event

PEA mouse(A5)
_GetMouse
LEA mouse(A5),A0 ;current mouse
LEA oldmouse(A5),A1 ;old mouse;find where mouse is
MOVE.W (A0),D0 ;new mouse
MOVE.W (A1),D1 ;old mouse
CMP D1,D0 ;has y mouse changed?
BEQ statck ;no, don't update paddle
;yes, make new paddle
JSR MOVEPADDLE ;update paddle position

statck:
JSR MOVEBALL ;advance ball
JSR BOUNCE ;change ball direction

RTS
; --- MOVE PADDLE SUBROUTINE ----

MOVEPADDLE:

; copy paddle position to oldpaddle

LEA paddle(A5), A0
LEA oldpaddle(A5), A1
MOVE.W (A0), (A1) ;top
MOVE.W 2(A0), 2(A1);left
MOVE.W 4(A0), 4(A1);bottom
MOVE.W 6(A0), 6(A1);right

; update paddle position

LEA mouse(A5),A0 ;get current mouse position
LEA paddle(A5),A1 ;update paddle position
MOVE.W (A0), (A1) ;update TOP
MOVE.W (A0),D0 ;calculate BOTTOM
ADD.W Paddlelength(A5),D0
MOVE.W D0,4(A1) ;update BOTTOM

```

; draw new paddle

```
PEA  paddle(A5)
PEA  PaddlePat
_FillRect
```

; calculate difference rectangle

```
LEA  paddle(A5),A0      ;current paddle
LEA  oldpaddle(A5),A1    ;old paddle
CLR.L D0
CLR.L D1
MOVE.W (A0), D0
MOVE.W (A1), D1
LEA  paddle(A5), A0
LEA  oldpaddle(A5), A1
SUB.WD1, D0
BMI  up
dn:  MOVE.W (A0), 4(A1) ;moved down
      BRA wipeout
up:  MOVE.W 4(A0), (A1) ;moved up
      BRA wipeout
```

wipeout:

```
PEA  oldpaddle(A5)
PEA  BackPat0
_FillRect
```

; update mouse

```
LEA  mouse(A5),A0      ;current mouse
LEA  oldmouse(A5),A1    ;old mouse
MOVE.L (A0),(A1)      ;update old mouse
RTS
```

; --- MOVE BALL SUBROUTINE ---
MOVEBALL:

; copy Ballrgn to oldballrgn

```
MOVE.L Ballrgn(A5),A0
MOVE.L A0,-(SP)
MOVE.L oldballrgn(A5),A0
MOVE.L A0,-(SP)
_CopyRgn
```

; move ball region

```
MOVE.L Ballrgn(A5),A0
MOVE.L A0,-(SP)
MOVE.W dh,-(SP)      ;horiz. increment
MOVE.W dv,-(SP)      ;vert. increment
_OffsetRgn           ;offset ball rgn
```

; draw new BALL

```
MOVE.L Ballrgn(A5),A0
MOVE.L A0,-(SP)
PEA  BallPat
_FillRgn
```

; erase old ball difference

```
MOVE.L Ballrgn(A5),A0
MOVE.L oldballrgn(A5),A1
MOVE.L unionrgn(A5),A3
MOVE.L A0,-(SP)
MOVE.L A1,-(SP)
MOVE.L A3,-(SP)
_Unionrgn
```

```
MOVE.L unionrgn(A5),A0
MOVE.L Ballrgn(A5),A1
MOVE.L unionrgn(A5),A3
MOVE.L A0,-(SP)
MOVE.L A1,-(SP)
MOVE.L A3,-(SP)
_Diffrgn
```

```
MOVE.L unionrgn(A5),A0
MOVE.L A0,-(SP)
PEA  BackPat0
_FillRgn
```

RTS

; ----- BOUNCE BALL -----
BOUNCE:

```
CLR.B -(SP)
PEA  Paddle(A5)          ;push rect
MOVE.L Ballrgn(A5), A0
MOVE.L A0, -(SP)        ;push rgn handle
_RectInRgn
MOVE.B (SP)+, D0
CMP  #0, D0              ;false?
BEQ  RETBOUNCE          ;yes...
```

```
MOVE.L Ballrgn(A5),A0 ;no...
MOVE.L A0,-(SP)      ;erase ball
PEA  BackPat0
_FillRgn
JSR INITBALL          ;start new ball
```

RETBOUNCE:
RTS

; ---- QDSTUFF SUBROUTINE ----
QDSTUFF:

; DRAW GAME FIELD

```
_PenNormal
PEA  BACKPAT0
_BackPat
PEA  Field.of.play(A5) ;frame rectangle
_EraseRect
PEA  Field.of.play(A5)
_FrameRect      ;draw game rectangle
```

; DRAW PADDLE

```

PEA Paddle(A5)
PEA PaddlePat
_FillRect
PEA Paddle(A5)
_FrameRect

RTS
; --- RETURN TO FINDER -----

EXIT: RTS ; return to finder

; ---LOCAL DATA AREA -----

WPOINTER: DC.L 0 ;store window pt
WBOUNDS: DC.W 40 ;rectangle top
          DC.W 2 ;left
          DC.W 335 ;bottom
          DC.W 508 ;right
WINDTITLE: DC.B 24 ; title length
           DC.B 'BALL AND PADDLE EXAMPLE ',0

```

EventRecord:

```

What: DC.W 0 ; what event
Message: DC.L 0 ; ptr. to msg
When: DC.L 0
Point: DC.L 0
Modify: DC.W 0

```

```

BackPat0: DC.L $AA55AA55 ;playing field
          DC.L $AA55AA55

```

```

BallPat: DC.L $00000000
         DC.L $00000000

```

```

PaddlePat: DC.L $FFFFFF
           DC.L $FFFFFF

```

```

dh: DC.W -1 ;BALL INCREMENTS
dv: DC.W 0

```

; ---- APPLICATION GLOBALS ----

```

Field.of.play: DS.W 1 ;frame size storage
               DS.W 1 ;for game field rectangle
               DS.W 1
               DS.W 1

```

```

Paddle: DS.W 1 ;frame size storage
         DS.W 1 ;for paddle rectangle
         DS.W 1
         DS.W 1

```

```

oldpaddle: DS.W 1 ;frame size storage
           DS.W 1 ;for paddle rectangle
           DS.W 1
           DS.W 1

```

```

Paddlelength: DS.W 1

```

```

Ball: DS.W 1 ;frame size storage

```

```

DS.W 1 ;for ball rectangle
DS.W 1
DS.W 1

```

```

Ballwide: DS.W 1
Ballhigh: DS.W 1

```

```

Ballrgn: DS.L 1 ;ball region handle
oldballrgn: DS.L 1 ;old ball region handle
unionrgn: DS.L 1 ;temp region

```

```

mouse: DS.L 1
oldmouse: DS.L 1

```

; ---- END OF PROGRAM ----

;ANIMATE_rscs.asm

; resource file for the animate example
; created using the assembler
; signature is creator tag
;

RESOURCE 'ANIM' 0 'IDENTIFICATION'

DC.B 30, 'animate example--Feb. 27, 1985'

.ALIGN 2

RESOURCE 'BNDL' 128 'BUNDLE'

```

DC.L 'ANIM' ;NAME OF SIGNATURE
DC.W 0,1 ;DATA (DOESN'T CHANGE)
DC.L 'ICN#' ;ICON MAPPINGS
DC.W 0 ;NUMBER OF MAPPINGS-1
DC.W 0,128 ;MAP 0 TO ICON 128

```

```

DC.L 'FREF' ;FREF MAPPINGS
DC.W 0 ;NUMBER OF MAPPINGS-1
DC.W 0,128 ;MAP 0 TO FREF 128

```

RESOURCE 'FREF' 128 'FREF 1'

DC.B 'APPL', 0, 0, 0

.ALIGN 2

RESOURCE 'ICN#' 128 'MY ICON'

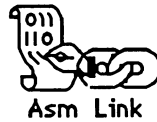
; FIRST APPLICATION ICON BIT MAP

INCLUDE animate.icon.asm



Assembly Lab

Jasik Hypes MacNosy



Steve Jasik
MacTutor Contributing Author
MacTutor Vol. 1 No. 6

A New Disassembler

As a compiler writer specializing in code generation, I decided to build a Disassembler to better understand the Motorola 68000 and the Macintosh operating system. Given the poor quality of the documentation and the general lack of source listings, a powerful disassembler is a useful tool for debugging and getting information about the system.

MacNosy and its helper programs consist of over 7000 lines of Pascal and about 300 lines of assembly language code. Internally, MacNosy contains many features usually associated with a compiler such as a table manager, symbol table enter/lookup routines, a reference map, global flow analysis, etc.

Features

MacNosy has many features that place it an order of magnitude above other disassemblers. They include:

- Symbol dictionaries of the Rom names and global symbols (0 - \$B00) along with value to symbol substitution in appropriate places.
- Selective list of procedures in a file by procedure name or substring.
- Ability to place the disassembled output on a file in assembler listing output or assembler input format (MDS ".list" or ".asm" format).
- References to the symbols are collected and may be selectively viewed.
- Ability to search the program file for references to selected addresses, trap (rom) calls, resource type references, constant or string references.
- Ability to translate the segment relative address of an instruction to the disk file relative address for code patching purposes.
- MacNosy records its input on a ".jrnl" file (in text format) for later playback. This feature is used as an educational tool and as a medium of communication between developers, hackers, etc.
- Ability to reformat data in its "natural format" via directives. This is in addition to the automatic recognition of various character string formats.
- A full or selective listing of the resources in a file. Format is similar to that of the Resource Mover, but you get more information with less work.
- A built-in mini editor to view files w/out leaving Nosy.

Facts and Specifications

MacNosy runs on a 512K Mac or a 1 Meg Lisa under the Workshop O.S. or the Macworks environment.

It is capable of disassembling the resource fork of any application file, ROM, Macsbug, and various resource types in the System file (DRVr, PACK, INIT, CDEF, WDEF, etc). Note that source listing of the WDEF, CDEF procedures come with the MacSupplement.

The released disk contains both the Mac and Lisa versions of MacNosy, some sample ".jrnl" files to acquaint you with it, source code for the table manager (this may be useful to other developers) and the SfGetFile routines so you can see how it selects files to disassemble.

How Does MacNosy Work?

When disassembling a file, one must know the structure (code or data) of a given piece of the file. To do this, Nosy uses the fact that every program has a "call graph". The nodes of the graph are the procedures and the edges are the calls (A calls B, etc). As Nosy is primarily interested in the size of the nodes, it ignores some aspects of the graph (such as recursion) which leads to cycles in the graph. Because of this, the graph reduces to a Directed Acyclic (without cycles) Graph. We can get sloppy in our terminology and refer to the DAG as a "tree of procedures".

Discovering the size and extent of a procedure is a messy problem that entails disassembling instructions, and looking for the procedure exits. The potential presence of spaghetti code makes the algorithm more complicated.

At this point a reasonable strategy is to treat the file as a collection of code and data blocks (contiguous set of bytes). We start at the entry points and walk the tree until all possible procedures have been discovered. The remaining areas are categorized as "data" blocks. During the tree walk, Nosy builds symbol tables for the various categories of labels which are:

Procedure (JSR X) X = proc'nnn' or the 8 character name following the proc if compiled by Lisa Pascal with the D+ option on.

Global labels (X(A5))

X = glob'nnn'

Data labels (LEA X or PEA X)

X = data'nnn'

Common labels (JMP X, Bcc X from another proc) - com_'nnn'

Local labels within a procedure - loc_'nnn'

You can change the names of all but the local labels in Nosy.

Example 1 - A fragment of the resource list of a System file. Note that for the DITL's, the accompanying text/controls are listed.

```

26 resource types, data index = 100
Type FRSV att indx length name
ID 1 00 CF12 A 0 0
Type ALERT att indx length name
ID -3997 20 CDBC C 0 0
ID -3996 20 CDCC C 0 0
ID -3995 20 CDDC C 0 0
ID -3994 20 CDEC C 0 0
Type DITL att indx length name
ID -15936 20 BA6C 1E
ID -6047 20 CA18 1A6 <Initialize> <Eject> <OK>
Do you want to initialize it?
This disk is unreadable:
This disk is damaged:
This is not a Macintosh disk:
Please name this disk:
Initializing disk . . .
Initialization failed!
Initialize this disk?
ID -4000 20 CC7C A8 <Open> <O> <Cancel> <Eject> <Drive> h
ID -3999 20 CD28 90 <Save> <Cancel> Save as: <Eject> <Drive>

```

Example 2 - code fragment from a little test program I wrote. The output listing format is:
aaa: hhhh hhhh 'cccc' label opcode address \$ssaaaaaa
aaa = segment relative address, hhhh and ccccc are the value of the instruction in hex and ascii. The field ssaaaaaa is the address of any label or symbol reference in the address field. ss is the segment number of the reference.

I have suppressed leading zeros in most cases and many of the numeric formats use decimal conversion. Macros POP and PUSH have the obvious meaning. The QUAL pseudo implies that all the labels of the form loc_nnn are local to the procedure. The line right after it tells us who calls it. The line with "_Button" is a trap macro. The DNAME macro expands to the 8 character name which is used by Lisabug and Nosy.

```

440:          QUAL  CHK_USER
          ; refs - TEST
440:
440: 4E56 0000 'NV..' CHK_USER LINK A6,#0
444: 4267 'Bg' CLR -(A7)
446: A974 '.t' _Button
448: 101F '..' POP.B D0
44A: 6714 'g.' BEQ loc_2 $1000460
44C: 2F0E '/.' PUSH.L A6
44E: 4EBA FFC0 'N...' JSR WAITBUTT $1000410
452: 4267 'Bg' loc_1 CLR -(A7)
454: A974 '.t' _Button
456: 101F '..' POP.B D0
458: 67F8 'g.' BEQ loc_1 $1000452
45A: 2F0E '/.' PUSH.L A6
45C: 4EBA FFB2 'N...' JSR WAITBUTT $1000410
460: 4E5E 'N^' loc_2 UNLK A6
462: 4E75 'Nu' RTS
464:

```

464: C348 data21 DNAME CHK_USER,0,2

Example 3 - This next routine was dumped to the hardcopy file in ".asm" format. It is suitable for input to the MDS assembler ASM.

```

QUAL  WCR
; refs - TEST

WCR  LINK  A6,#0
      PUSH.L glob13(A5)
      PEA   data19
      CLR   -(A7)
      JSR   %W_STR
      PUSH.L glob13(A5)
      JSR   %W_LN
      PUSH.L glob12(A5)
      PEA   glob8(A5)
      PUSH  #255
      JSR   %R_STR
      PUSH.L glob12(A5)
      JSR   %R_LN
      UNLK  A6
      RTS

```

data18 DNAME WCR ,0,2

; refs - WCR+8

data19 STR 'wait for cr'

Example 4 - Note Value to Symbol substitution in this ROM fragment listing.

```

404C12: 50F8 0902      'P...' Launch S_T      LaunchFlag      $902
404C16: 31E8 0004 0936 '1....6 loc_2      MOVE 4(A0),CurPageOption $936
404C1C: 2058           'X'           MOVE.L (A0)+,A0
404C1E: 43F8 0910      'C...'      LEA CurApName,A1      $910
404C22: 7020           'p'           MOVEQ #32,D0
404C24: A02E           '..'          _BlockMove

```

Example 5 - Sample Reference Map listing fragment of the "System Globals" from ROM

```

114 HeapEnd      proc203 MaxMem proc253
118 TheZone      proc201 proc203 GetZone MaxMem proc212 proc232
                  InitResources proc852 MoreMasters InitZone proc204
                  SetGrowZone
11C uTableBase  proc36 proc89 proc96 RDrvrInstall SystemTask proc942
                  SystemMenu OpenDeskacc BlockMove com_28

```

Example 6 - Sample Segment Reference Map listing fragment of Nosy. Inter-seg refs are prefixed with "n/". This map may be used for procedure balancing.

```

seg# procedure  fba blen refs (seg#/proc) called by
1] HEAP_OVF    1D6  50 ADD_USED  4/NEW_TBL
1] SET_MAX     208 120 CLR_TBL   4/REL_TBLS 3/RTN_PTR
1] CLR_TBL     280  46 DI_PROC   3/SEARCH_C 4/DI_FILE 3/SEARCH_C
1] MOVEUP      2AE  80 ADD_USED
1] ADD_ENTR    2FE  42 DI_PROC   ENTER_LA PUT_REF NEW_CASE
                  ADD_ISPR 3/CHK_PROC 4/NI_FILE 4/DI_FILE
                  PROCESS_
1] SET_USED    328  44 4/NI_FILE

```

Disassembling the ROM

I would like to show a sample listing of a piece or two of ROM but I don't want to start any fights with Apple's Lawyers. So like sex, I leave it for you to do it in the privacy of your own home.

I found a few interesting things looking around the ROM. One is an interesting piece of unreachable code at 40AD30 which blasts 32 long words into RAM and then hangs. Another is the "come from" code in the rom patch area in the system heap. In many cases bugs were patched by placing a CMPI.L \$40xxxx,28(A7) followed by a suitable jump in unrelated routines. A rather obnoxious example is BlockMove which contains 3 such checks. I will be forming a MacNosy Users Group (Special Interest Group) on the Delphi Information Service (800-544-4005) starting in April to study the ROM and swap "jrn1" files.



Assembly Lab

A Micro-Finder Utility



Chris Yerga
Contributing Editor
MacTutor Vol. 1 No. 7

A few weeks ago I was compiling a disk of my favorite games for a friend of mine. Disk space being as valuable as it is, the Finder's use of close to 50K of disk space seemed a terrible waste on a disk containing files that would probably never be manipulated in any way. Armed with my copy of *Inside Macintosh*, I set out to correct this injustice. My goal was to write a short program that would present the user with a list of the available applications and allow him to execute the one of his choosing. As is usually the case when I set out to write a program, I found that what seems at first to be an arduous task (up to this point I had never tried to access the disk drive and the file directory), is actually quite simple when full advantage is taken of the Mac's built in routines. This article will concern itself with two such routines: the Standard File routine and the Launch facility.

Working according to the flow of the program, our first concern is to list the available applications and allow the user to select the one he wants to execute. Anyone who has ever opened a document from an application should be familiar with the Standard Get File dialog box [See BASIC column for an introduction to Standard File dialog box]. This standard method of opening a file is quite simple to implement through the convenience of the Package Manager. The package manager allows an application to call one of several "packages" - short specialized routines kept in the System file - from its main code. This saves the programmer from having to write code to perform the tasks that are provided for by the Package Manager. There are packages for opening files, saving files, initializing disks, and performing Binary/Hex conversion among others.

The package that we are going to use is called the Standard Get File package. It lists all the files of a certain filetype that are on a disk and allows the user to choose one of them. As programmers, virtually all that we are required to do is tell the package manager what filetype(s) that we want listed and where to put the resulting information. The PM takes care of the rest: it creates the dialog box, reads the disk directory, handles events, and stores the input that it gets from the user. This is exactly what we want, so our task is simplified immensely by this routine. However, if we did need the PM to function a bit differently, it has provisions for designing custom dialog boxes and performing special event handling tasks.

Page 13 of the Package Manager Programmer's Guide portion of *Inside Macintosh* describes the standard get file procedure's PASCAL interface as follows:

Procedure SGetFile (**where**: Point; **prompt**: Str255;
fileFilter: ProcPtr; **numTypes**: INTEGER; **typeList**:
SFListPtr; **dlgHook**: ProcPtr; **VAR** **reply**: SFReply);

From assembly language, the first value that we need to push is **where**, which describes the top lefthand point at which the dialog box will be drawn. A bit of experimentation convinced me that (80,86) would serve our purpose well. The next variable, **prompt**, is not used by the current PM; however, older versions did require it to be passed, and so the PM accepts it to insure compatibility with older programs. **FileFilter** is used by applications that want to determine for themselves which files to display in the dialog box. We will pass NIL for this variable, because we would rather have the PM do it for us. **NumTypes** tells the PM how many different filetypes we want displayed. In our case this value is 1, since we only want applications displayed. Next, we push **typeList**, a pointer to the list of filetypes to be displayed. **DlgHook** is used by applications that need to draw a custom dialog box. Again, since the standard dialog box suits our needs, we will pass NIL. Finally, **VAR SFReply** points to the standard file reply record, a data structure through which the PM communicates with the calling application. The assembly language implementation looks like this:

SGetFile From Assembly

MOVE.W	#86,-(SP)	;Y coord. of where
MOVE.W	#80,-(SP)	;X coord. of where
PEA	'Prompt'	;prompt: unused, but ;we need to pass a ;string to keep the ;Package Manager ;happy
CLR.L	-(SP)	;fileFilter: unused, ;so we pass NIL
MOVE.W	#1,-(SP)	;numTypes: there is ;only 1 filetype in ;our list
PEA	TypeList	;typeList: points to ;our list of filetypes
CLR.L	-(SP)	;dlgHook: unused, so ;we pass NIL
PEA	SFReply	;SFReply: points to ;the Reply Record

The reply record contains information such as the filename, filetype, the volume where the file can be found, etc. Its structure, as described in *Inside Macintosh* is shown in figure 1.

The first variable, **good**, tells whether or not valid data is in the reply record. If it is FALSE, then the user hit the cancel button or for some other reason, the data should be ignored. If TRUE, then the call was successful and the calling application can process the data. Copy is unused;

PASCAL Structure		Assembly Equivalent
SFReply = RECORD		
good:	BOOLEAN;	{ 1 Byte - DC.B 0 }
copy:	BOOLEAN;	{ 1 Byte - DC.B 0 }
fType:	OSType;	{ 4 Bytes - DC.B 'TYPE' }
vRefNum:	INTEGER;	{ 2 Bytes - DC.W 0 }
version:	INTEGER;	{ 2 Bytes - DC.W 0 }
fName:	STRING[63]	{ 63 Bytes - DCB.B 63,0 }
END;		

Fig. 1 Reply record from Standard File

Inside Macintosh gives no explanation. The next variable is **fType**, the 4 character filetype of the file selected. We do not need to look at this, because it will always be 'APPL' -- the filetype for applications. **VRefNum** is the volume reference number of the volume that contains the selected file. This is needed in many file manager calls. **Version** is the version number of the selected file. Perhaps the most useful piece of information is **fName**, the filename of the selected file.

After everything has been set up, we are ready to invoke the standard get file package as follows:

```
MOVE.W    #2,-(SP)    ;Call SGetFile through
_Pack3                      ;the package manager
```

After checking **good** and determining that a successful call has occurred, the only remaining task is running the selected application.

The routine responsible for running an application is located in the Segment Loader and is called Launch. The launch routine is unique in the method used to call it. Unlike the toolbox traps, in which values are passed on the stack, launch is a routine whose values must be pointed to by an address register. Register-based routines such as this are common in the low level system portion of the Macintosh ROM.

In calling the launch routine, the programmer passes a pointer to the Launch Pointer Record in A0 [Fig 2]. The first 4 bytes of the LP record (no pun intended...) consist of a pointer to the filename of the application we want to run. As all Macintosh programmers must quickly learn, the assembly language equivalent of a pointer is an Effective Address. In order to get a pointer to the filename, we load its effective address into address register A1 by the instruction LEA

Launch Pointer Record Structure

```
LaunchPtr: DC.L    0    ;Pointer to Filename of application
           DC.W    0    ;Memory allocation ( 0 = Standard )
```

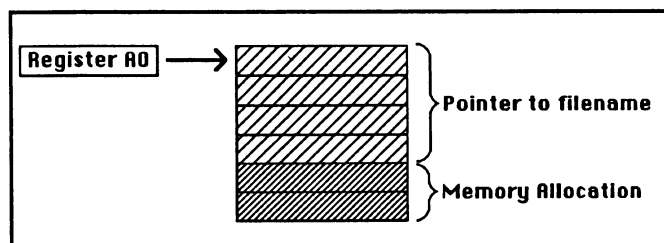


Figure 2: The Launch Pointer Record

FileName,A1. When we have the effective address (or pointer) in a register, we are free to store it in the proper location of the LP record. The last 2 bytes compose a word of data that tells the launch routine how we want memory allocated to the application. A value of NIL in this location gives standard memory allocation (as the Finder would), so we do not need to change it.

After the LP record has been set up and A0 is pointing to it, the trap _Launch is called and we are finished.

The balance of the program consists of initialization calls to the various managers and a few QuickDraw traps to display the title of the program. The program as I have implemented it is functional but rather spartan in its lack of features and appearance. No resources were included and there is no use of menus or windows. The purpose of this article was to describe some interesting parts of the toolbox, not to reiterate the material that David has described in his column. However, I encourage you to liven your own versions up by putting some graphics or a description of the program somewhere on the screen. At least put your name in it and impress your friends!

```
;🍏 MicroFinder
```

```
;🍏
```

```
;🍏 A startup program that allows
```

```
;🍏 the user to select the
```

```
;🍏 the application that he wants
```

```
;🍏 to run of those on the disk.
```

```
;🍏 June 1985.
```

```
;🍏
```

```
;🍏 © MacTutor by Chris Yerga
```

```
;🍏 🍏 Contributing Editor
```

```
;🍏 ToolBox & System Traps
```

```
_TRAP      _InitCursor      $A850
_TRAP      _InitGraf        $A86E
_TRAP      _SetPort         $A873
_TRAP      _BackPat         $A87C
_TRAP      _DrawString      $A884
_TRAP      _TextFont        $A887
_TRAP      _TextSize        $A88A
_TRAP      _MoveTo          $A893
_TRAP      _PenPat          $A89D
_TRAP      _FrameRect       $A8A1
_TRAP      _PaintRect       $A8A2
_TRAP      _EraseRect       $A8A3
_TRAP      _InitFonts       $A8FE
_TRAP      _GetWMgrPort     $A910
_TRAP      _InitWindows     $A912
_TRAP      _InitMenus       $A930
_TRAP      _InitDialogs     $A97B
_TRAP      _TEInit          $A9CC
_TRAP      _FlushEvents     $A050
_TRAP      _InitPack        $A9E5
_TRAP      _Pack3           $A9EA
_TRAP      _Launch          $A9F2
```

```
;🍏 Declaration of external labels
```


XDEF START ;For the linker

⌘ Local Constants ⌘

```
AllEvents equ $0000FFFF
Athens equ 7 ;Our text values
AthenSize equ 18
```

⌘ Start of Main Program ⌘

START:

```
MOVEM.L D0-D7/A0-A6,-(SP)
LEA SAVEREGS,A0
MOVE.L A6,(A0)
MOVE.L A7,4(A0)
```

⌘ Initialize the ROM routines ⌘

```
PEA -4(A5) ;QD Global ptr
_InitGraf ;Init QD global
_InitFonts ;Init font manager
_InitWindows ;Init Window Mgr
_InitMenus ;Init menu manager
```

```
CLR.L -(SP) ;Get standard
;failsafe procedure
_InitDialogs ;Init Dialog Manger
_TELnit ;Init Text edit for
;the heck of it
```

```
MOVE.W #2,-(SP)
_InitPack ;Init Package Mgr
```

```
MOVE.L #AllEvents,D0 ;And flush
_FlushEvents ;events
_InitCursor ;Get the arrow
```

⌘ Start of the Main Code ⌘

```
PEA Gptr ;Get Handle to WMgrPort
_GetWmgrPort
```

```
LEA Gptr,A0
MOVE.L (A0),-(SP) ;Push address of
;the ptr to the port
_SetPort ;And open the port
```

```
PEA GrayPat ;Set the Background
;Pat'rn to 50% Gray
```

```
_BackPat
PEA Screen ;And clear the screen
_EraseRect
```

⌘ Display the Title of the program ⌘

```
PEA WhitePat ;white rectangle text
_PenPat ;Set the pattern to white
PEA TitleRect ;Point to our rectangle
_PaintRect ;and fill it with the Pen Pat.
```

```
PEA BlackPat ;draw a thin Black border
_PenPat ;Set the pattern to black
PEA TitleRect ;Point to our rectangle
_FrameRect ;and draw the frame of it
```

```
MOVE.W #135,-(SP) ;position Pen
MOVE.W #60,-(SP) ;Screen (60,135)
_MoveTo ;Position pen...
MOVE.W #Athens,-(SP) ;Choose font
_TextFont ;(Athens = 7)
MOVE.W #AthenSize,-(SP) ;size of text
_TextSize ;(AthenSize = 18)
```

```
PEA 'LaunchAp 1.0 by Chris Yerga'
_DrawString ;And draw it...
PEA WhitePat ;Background Pat. to white
_BackPat ;So the dialog box looks
;normal...
```

MainLoop:

⌘ SFGETFILE ROUTINE ⌘

```
MOVE.W #86,-(SP) ;Y Coordinate
MOVE.W #80,-(SP) ;X Coordinate
PEA 'PROMPT' ;Prompt (Unused)
CLR.L -(SP) ;NIL for FileFilter
MOVE.W #1,-(SP) ;We only want 1 file
;type listed
```

```
PEA TypeList ;Point to the Type
;List
CLR.L -(SP) ;NIL for dlgHook
PEA SFReply ;Point to the Reply
;Record
MOVE.W #2,-(SP) ;Routine Selector
;for SFGetFile
```

```
_Pack3
LEA SFReply,A0 ;Get the result
;from the Package
;Manager
CMP.B #0,(A0) ;was it successful
;(good = TRUE) ?
```

```
BEQ MainLoop ;Nope, user hit the
;cancel button, try
;again till he
;figures it out
PEA BlackPat ;Clear the screen
;to Black
```

```
_PenPat
PEA Screen
_PaintRect
```

⌘ Launch Routine ⌘

```
LEA SFileName,A1 ;Get the address of
;the filename
LEA LaunchPtr,A0 ;Get the address of
;the Launch Ptr in
;A0 (according to
;Register Based
;conventions.
```

```

MOVE.L    A1,(A0)      ;Move the Ptr to the
                        ;address of the
                        ;filename (in A1)
                        ;to the first word
                        ;of the Launch Ptr
_launch    ;And call Launch
                        ;...zoom!

;🍏                                     🍏
;🍏 The Program's Variables             🍏
;🍏                                     🍏

SAVEREGS: DCB.L 2,0

GPTR:     DS.L 1        ;Space for
                        ;the GrafPtr

SCREEN:    DC.W 0,0,342,512
                        ;Rectangle describing the full
                        ;Macintosh screen

TITLERECT: DC.W 41,127,69,381
                        ;Rectangle that contains the name
                        ;of the program etc.

TYPELIST:  DC.L 'APPL'  ;The list of file
                        ;types we want
                        ;displayed in the
                        ;Standard File
                        ;dialog box. In this
                        ;case, there is
                        ;only 1 type

;🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏
;🍏 The Standard File Reply Record
;🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏🍏

SFREPLY:   DC.B 0,0      ;good, copy (BOOLEAN)
           DC.B 'TYPE'   ;fType (OSType)
           DC.W 0,0      ;vRefNum,version(INT)
SFILNAME:  DCB.B 63,0    ;fName (STRING[63])

;🍏 The Launch Pointer Record 🍏

LAUNCHPTR: DC.L 0        ;Ptr to the
                        ;FileName
           DC.W 0        ;Mode (0 =
                        ;standard
                        ;memory
                        ;allocation)

;🍏 Pattern Definitions 🍏

GRAYPAT:   DC.B $55,$AA,$55,$AA,$55,$AA,$55,$AA
BLACKPAT:  DC.B $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
WHITEPAT:  DC.B 0,0,0,0,0,0,0,0

```

;End of Program Source Code

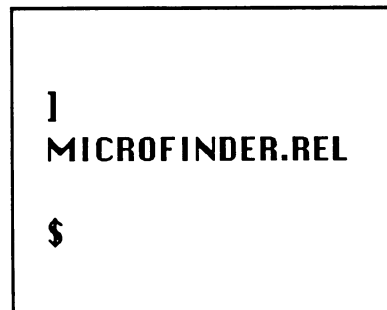


Fig. 3 Linker File

Editor's Note:

There is a subtle bug in this program that causes an error 26 when launching from the external disk drive. Letters were requested to fix this problem and the following shows the necessary fix. *David E. Smith*

MicroFinder Bug Fix Shawn Mikiten San Antonio, Tx.

MicroFinder [June MacTutor 1-7] works well as long as you do not select a file on the other disk drive. The bug appears in the way the _Launch trap runs the application. The filename is properly passed but not the volume the file is on. The _Launch trap will automatically attempt to run the application on the disk the MicroFinder was started from because the filename did not have the volume specifier before it. The solution lies in the _Pack3 reply record. The variable vRefNum contains the disk drive number of the file when it returns from a successful call. Using the vRefNum value obtained, it is put in a parameter block for the File Manager call _SetVol. The parameter used is ioDrvNum. The following are directions for patching the source code so that it will work (See File Manager /OS/FS.A.1, page 26 & 33 in the new "Inside Macintosh" phone book.)
Add _SetVol trap value:

```
.TRAP  _SetVol      $A015
```

The code following _PaintRect is to be added:

```
_PaintRect
```

```
; Launch Routine here
```

```

LEA     PARMBLK, A0    ;get pointer
LEA     vRefNum, A1    ; get addr. of vRefNum
MOVE.W  (A1), 22(A0)   ; put vRefNum into the
                        ; ioDrvNum parameter.
_SetVol                                ; set it. (note A0 points to
                        ; the param. block!)

```

LEA SFileName, A1 ; get addr. of filename

Next, add the data structures necessary for the Reply record and parameter block. Add vRefNum in the Reply record:

```
SFREPLY:     DC.B    0,0        ; good, copy
              DC.B    'TYPE'   ; FType (ostype)
vRefNum:     DC.W    0,0        ; vRefNum, version
SFILENAME:   DCB.B   63,0       ; fname (string[63])
```

And now the parameter block, to be added after the WHITEPAT definition (Note that the file name pointer must be nil to force invocation of vRefNum inserted into ioDrvNum.):

; standard 8 field parameter block for setvol.

```
PARMBLK:     DC.L    0        ;ioLink ptr.
              DC.W    0        ;ioType
              DC.W    0        ;ioTrap
              DC.W    0        ;ioCmdAddr
              DC.L    0        ;ioCompletion ptr.
              DC.W    0        ;ioResult
              DC.L    0        ;ioFileName ptr.
              DC.W    0        ;ioDrvNum
```

I hope this bug fix will offset the cost of the MDS software I am about to purchase. I have been using a friend's copy and find it the easiest assembler I have used in 7 years of Apple II's, Digital 2065, IBM PC and VAX computers!

[Shawn wins the \$50 for being first with his bug fix. Thank you to the four other people who submitted a fix: Loftus Becker Jr. of Hartford, CT, who submitted a similar solution to Shawn's; Tom Taylor of Provo Utah and Neal Lebedin of Palm Bay, FL., both of whom submitted similar solutions but slightly different from Shawn's; And Paul Snively of Columbus, IN. in MacAsm...-Ed.]

More Bug Fix Comments
Loftus E. Becker, Jr.
Hartford, CT.

The bug in Chris Yerga's microfinder program (ID=26) is "failure to launch". This stems from the fact that the launch routine will look for the named file on the default volume unless it is told otherwise. Hence, when the microfinder program is run from the default volume (usually drive 1) and tries to launch a program on drive 2, Launch gets the filename, doesn't know it's on drive 2, and tries to launch from drive 1!

Two points may be of some interest. First this is a bomb that is not trapped by Macsbug. Second, a similar bug appears in the MDS development system: If you are developing a program with the .asm, .link, and other files on the external drive, but the program itself on the internal drive, an attempt to run it from the TRANSFER menu in the linker will produce the same error. My compliments on a magazine that has gotten better with every issue.

Alternate Bug Fix
Tom Taylor
Provo, UT

This is similar in function, but uses the stack for the parameter block...

```
ioVQEISize    EQU   $40    ;io param blk length
ioCompletion   EQU   $C    ;offset (ptr.)
ioVNPtr       EQU   $12    ;offset (ptr. or nil)
ioVRefNum     EQU   $16    ;offset (word)
```

```
SUB.L   #ioVQEISize, SP    ;allocate block on stack
MOVE.L   SP, A0            ;block ptr. to A0
CLR.L   ioCompletion(A0)   ;I always clear this
LEA     SFReply, A1        ;reply record ptr to A0
MOVE.W   6(A1), ioVRefNum(A0)
CLR.L   ioVNPtr(A0)        ;clear name ptr.
```

```
_SetVol                    ; go for it...
ADD.L   #ioVQEISize, SP    ; dump block
```

MacAsm Version of Bug Fix
Paul Snively
Columbus, IN

Apparently the standard file package, even though it automatically senses the presence of a second disk drive and, if one is present, gives you the "Drive" button in the standard dialog box, *does not automatically change the default volume if the "Drive" button is hit.* Where I come from, features like this are called "bugs." In other words as far as I'm concerned, the bug is in the standard file package, not Chris' code

```
01150 *-----
01160 *     Launch Routine (MacAsm)
01170 *-----
01180     LEA   ParamBlock(PC),A0 ;File Man. param Blk
01190     LEA   vRefNum(PC),A1    ;Volume Ref Number
01200     MOVE.W (A1),22(A0)     ;Move to paramBlock
01210     OST   SetVol            ;Make the default volume
01220     LEA   SFileName(PC),A1 ;Get addr of file name
01230     LEA   LaunchPtr(PC),A0 ;Get addr of launch ptr
01240     MOVE.L A1,(A0)         ;Move pointer to name
01250     TBX   Launch            ;And call Launch...
```



Advanced Mac'ing

Resource Formats in Assembly



Frank Alviani
Waukegan, Ill.
MacTutor Vol. 1 No. 7

Resources are the most mystifying of all the Mac technology, mainly because they are undocumented, and not usually found on other computers. The purpose of resources are to separate text, icons, fonts, menu items and dialogs from the executable source code of a program, where they can be easily changed without re-compilation. With the resource editor and other tools, this opens up a world of "user customized" programs. The user literally changes the program resources to his own tastes. This makes for an exciting possibility in software adaptation that has yet to be fully explored. In the meantime, programmers still must struggle to make use of resources effectively.

A big problem is not having adequate documentation of the format of each resource in terms of a binary file specification; exactly how many bytes and in what format each resource item is specified. While the RMaker provides an easy way to assembly resources, it does little to explain their internal format. The Apple Assembler has the ability to compile resources directly, offering an alternative to the RMaker utility. All that is lacking is the correct assembler source code format of each resource. Now Frank Alviani has filled this gap by providing us with the assembler format of all the major resources. This information should go a long way toward improving the resource void. We encourage others to help in this area by sending in articles or information on resources to help improve our understanding of this vital Mac concept. [David E. Smith, Editor]

Linker or RMaker

The big advantage to using the linker rather than the RMaker is flexibility. You can control every detail - type style, etc. To balance this, linking is significantly longer and certain RMaker capabilities simply don't exist - "linking" in picture resources, for example. With RMover, those disadvantages are pretty small, however.

Only the reasonably common resources are documented here (mostly those in volume 1 of IM). I didn't include the Font resource defs, since nobody in their right mind would do a font in assembler (I had to do 2 in assembler for a DEC Pro system when I wrote a custom screen manager and I KNOW how bad that is!!). While there are probably others that aren't included, I've had no use for them.

The few notes regard (1) The correction to the "control type" field in the Control resource, and (2) an explanation of the enabling mask field in the Menu resource. These are naturally confusing since IM is wrong or silent on the matters.

I decided to take this approach to building a "prototypes"

file, rather than use macros, because (a) A typical macro call would be too big to fit on a line, and I am somewhat fussy about the appearance of my code (I have enough trouble without messy code confusing me further...) (b) I include explanatory comments on each line, so I don't have to keep referring to reference material for each field. Disk space is virtually free, so why not?

```
;
; Resource Prototypes for Assembler usage
;
; Frank Alviani
;
; More verbose than macros, but easier to copy & fill in...
;
; NOTES -
;
; ALL resources using local labels (labels starting with @)
; must be bracketed by regular labels; local labels can be
; re-used only if the duplicates are separated by regular
; labels. This applies even if no warning appears in the
; resource prototype!
; These are arranged roughly alphabetically, altho the
; DLOG/ALRT/DITL are grouped together at the end.
;
; In general # represents a numeric field normally filled in.
; The name and attribute fields on the RESOURCE line are
; optional.
; The idea is to have this in one editor window, and to copy
; to the resource file being built as needed. This will
; hopefully speed the process, and give flexibility the
; RMaker doesn't.
; There are some additional explanatory comments with
; various resources. Some of this is copied from
; MacTutor 1/4.
```

;RESOURCE ATTRIBUTES

SYSREF	EQU	128
SYSHEAP	EQU	64
PURGABLE	EQU	32
LOCKED	EQU	16
PROTECTED	EQU	8
PRELOAD	EQU	4
CHANGED	EQU	2

;RESOURCE FILE ATTRIBUTES

READONLY	EQU	128
COMPACT	EQU	64
MAPCHANGED	EQU	32

;DITL TYPE EQUATES

```

BUTTON    EQU    4
CHKBOX    EQU    5
RADIO     EQU    6
RESCTL    EQU    7
;
STATEXT   EQU    8
EDITEXT   EQU    16
ICONITM   EQU    32
QDPICITM EQU    64
USERITM   EQU    0
;
DISABLE   EQU    128

```

;WINDOW TYPES

```

DOCBOX    EQU    0    ;standard document window
ALERT     EQU    1    ;alert
PLAIN     EQU    2    ;plain
PLAINSHD  EQU    3    ;plain with shadow
NOGROWDOC EQU    4    ;doc window w/o grow box
ROUNDBOX  EQU    16   ;rounded-corner window
                ;(see IM for setting corner radius..)

```

;CONTROL TYPES

```

CBUTTON   EQU    0    ;simple button
CCHKBOX   EQU    1    ;check box
CRADIO    EQU    2    ;radio button
USEWFNT   EQU    8    ;add to above to use window's
                ;fonts
CSCROLL   EQU    16   ;scroll bar

```

; IDENTIFICATION resource - needed for Finder to locate icon

```

RESOURCE   'WCA1' 0 'IDENTIFICATION'
DC.B       AA1-@1
@1:        DC.B    'Ver. 0.1 3/31/85'
AA1:

```

;BUNDLE resource

```

.ALIGN     2
RESOURCE    'BNDL' ### 'name' [(attr)]
DC.L       'WCA1'    ;signature
DC.W       0,1       ;data (doesn't change)
DC.L       'ICN#'    ;icon mappings
DC.W       0         ;number of mappings - 1
DC.W       0,128     ;map 0 to icon 128

DC.L       'FREF'    ;FREF mappings
DC.W       0         ;number of mappings - 1
DC.W       0,128     ;map 0 to fref 128

```

;CONTROL resource

```

;
; NOTE - although the assembler definition of this claims
; that the "control type" field is a long word, IT LIES.
; Control type is a 16-bit field!!

```

```

.ALIGN     2
RESOURCE    'CNTL' ### 'name' [(attr)]
DC.W       #         ;top
DC.W       #         ;left
DC.W       #         ;bottom
DC.W       #         ;right

```

```

DC.W       #         ;initial value
DC.W       0         ;visible (T/F)
DC.W       #         ;max value
DC.W       #         ;min value
DC.W       #         ;control type
DC.L       0         ;refCon
DC.B       @2-@1     ;title length (at least 1)
@1:        DC.B      'xxx' ;title
@2:

```

;CURSOR resource

```

.ALIGN     2
RESOURCE    'CURS' ### 'name' [(attr)]
DC.L       #,#       ;1st 8 bytes of cursor
DC.L       #,#       ;2nd 8 bytes
DC.L       #,#       ;3rd 8 bytes
DC.L       #,#       ;4th 8 bytes
DC.L       #,#       ;1st 8 bytes of mask
DC.L       #,#       ;2nd 8 bytes
DC.L       #,#       ;3rd 8 bytes
DC.L       #,#       ;4th 8 bytes

DC.W       #,#       ;h,v of hot spot

```

;FREF resource

```

RESOURCE    'FREF' ### 'name' [(attr)]
DC.B        'APPL',0,0,0

```

;ICN# resource

```

.ALIGN     2
RESOURCE    'ICN#' ### 'name' [(attr)]
;there'd usually be an 'include' here...

```

;MENU resource

```

;
; NOTE - the "enable field": bits are number right to left 0-15.
; Bit 1 is the first menu item. A "0" bit in the mask disables
; that item. I haven't tried it, but I think turning off bit 0 of the
; mask disables the entire menu..

```

```

.ALIGN     2
RESOURCE    'MENU' ### 'name' [(attr)]
lbl:        DC.W      1    ;MENU ID
DC.W       0         ;width holder
DC.W       0         ;height holder
DC.L       0         ;std menu pro holder
DC.L       $1FF      ;enable all items
DC.B       @2-@1     ;title length (in bytes)
@1:        DC.B      20   ;title (this is the apple)
@2:

```

;MENU ITEM resource

```

DC.B       #         ;item length
DC.B       'xxx'     ;menu item
DC.B       0         ;no icon
DC.B       0         ;keyboard equivalent
DC.B       0         ;marking character
DC.B       0         ;style of item's text

DC.B       0         ;END OF MENU ITEMS

```

```

;PATTERN stuff
.ALIGN      2
RESOURCE    'PAT' ### 'name' [(attr)]
            DC.L      #,# ;1st, 2nd 4 bytes of pattern

.ALIGN      2
RESOURCE    'PAT#' ### 'name' [(attr)]
            DC.W      # ;# of patterns
            DC.L      #,# ;1st, 2nd 4 bytes of pattern #1

;STRING resource
.ALIGN      2
RESOURCE    'STR' ### 'name' [(attr)]
            DC.B      @2-@1 ;text length
@1:         DC.B      'xxx' ;text
@2:

;STRING LIST resource
.ALIGN      2
RESOURCE    'STR#' ### 'name' [(attr)]
            DC.W      # ;count of strings
            DC.B      @2-@1 ;text length - string #1
@1:         DC.B      'xxx' ;text
@2:
label:      ;REQUIRED REGULAR LABEL HERE!

;WINDOW resource
.ALIGN      2
RESOURCE    'WIND' ### 'name' [(attr)]
            DC.W      # ;top
            DC.W      # ;left
            DC.W      # ;bottom
            DC.W      # ;right
            DC.W      # ;window type
            DC.W      # ;visible (T/F)
            DC.W      # ;draw goAway (T/F)
            DC.L      0 ;refCon (available)
            DC.B      @2-@1 ;title length
@1:         DC.B      'xxx' ;title
@2:

; --- Dialog / Alert / DITL are grouped together ---

;DIALOG resource
.ALIGN      2
RESOURCE    'DLOG' ### 'name' [(attr)]
            DC.W      # ;top
            DC.W      # ;left
            DC.W      # ;bottom
            DC.W      # ;right
            DC.W      # ;window type
            DC.B      # ;visible (T/F)
            DC.B      0 ;IGNORED
            DC.B      # ;goAway flag (T=has close box)
            DC.B      0 ;IGNORED
            DC.L      0 ;refCon
            DC.W      # ;ID of DITL list

            DC.B      @2-@1 ;text length
@1:         DC.B      'xxx' ;text
@2:

```

```

;ALERT resource
.ALIGN      2
RESOURCE    'ALRT' ### 'name' [(attr)]
            DC.W      # ;top
            DC.W      # ;left
            DC.W      # ;bottom
            DC.W      # ;right
            DC.W      # ;resource ID of DITL list
            DC.W      # ;stages (see IM for details..)

;DITL resource
.ALIGN      2
RESOURCE    'DITL' ### 'name' [(attr)]
            DC.W      # ;# of items - 1

lbl:        DC.L      0 ;handle holder
            DC.W      # ;top
            DC.W      # ;left
            DC.W      # ;bottom
            DC.W      # ;right
            DC.B      type ;item-type
            DC.B      @2-@1 ;item length (MUST BE
EVEN)
@1:         DC.B      'xxx' ;item
@2:

; items must be even length
; item types are as follows
; control item - control-item + 4
; button 0
; check box 1
; radio button 2
; resource 3
;
; static text 8
; edit text 16
; icon item 32
; quickdraw pict 64
; user item 0
;
; disable item - item + 128
; system icons
; stop 0
; note 1
; alert 2

```



Mac Meets Ma Bell

A Communications Primer

Bruce Lieberman
MacTutor Contributing Editor
MacTutor Vol. 1 No. 8

Welcome to the wild and wacky world of Telecommunications. This article is not meant to be the definitive introduction to this subject (as it covers far too vast an assortment of related areas), just a bit of dipping your toes in the water and getting familiar with the terminology commonly in use.

In a nutshell, you are going to try and have your computer communicate a stream of binary digits to another computer, have everyone understand the process involved and make sure that the information got to its destination in an unadulterated form. There is a catch: you will be doing this across a very restricted (slow, limited frequency bandwidth), noisy and error prone medium: the telephone companies' phone lines and switching centers.

There's the rub. For speech, phone lines are good enough to maintain intelligibility (most of the time). For music, forget it, and for a stream of digital data...no way!

Enter the MODEM (MOdulate / DEModulate). This little goodie can take your digital data bits and convert them to an audio tone representing 1 or 0, then convert them back again. The frequency of these tones is predetermined by an accepted standard for the person originating or answering the call (Bell 103, Bell 212, etc.), and the tone frequencies themselves easily fit within the limitation bandwidth of the phone system.

In the old days, there were acoustically coupled modems that placed the old style phone handset into a box that had two holes surrounded with rubber (to make a tight seal around the mouthpiece and earpiece), within each hole was a complementary microphone and speaker to send and receive to and from the phone hand set. This approach had many drawbacks, not the least of which was induced physical noise. Nowadays we have direct connect modems that connect electronically to the phone lines and bypass the entire step of noise-producing acoustical to electrical conversion. Direct Connect modems have been taken a step further, to what is now termed "Intelligent" modems (most of which are based on the Hayes Smartmodem, which uses a standard licensed from Bizcomp, Inc.) that can perform a number of functions without the need of software; although all of these modems utilize software to make their functionality that much more flexible and approachable.

Now that we have a way of handling the limited frequency response of the phone lines, what do we do about all the noise and garbage on the line? The loss of one character of information can corrupt an entire file (program or text), so we must provide a way of dealing with the errors that are part and parcel of the phone system. Basically, all we can do is contrive a way to determine if the data survived the transmission intact, and if it has not, send it again until it is

correct (if there is a severe problem, the system will "time-out" after a predetermined number of retries). Each of the error correction protocols you will encounter will have its own way of accomplishing this task. The only thing you must keep in mind is that you have to run the same transfer protocol on each side of the transmission. We will take a quick look at some of the communications software packages now becoming available for the Macintosh, and following that I have included a glossary of telecommunications terms.

As you get started contacting the many public and private communications services available (BBS's, Dow Jones, Compuserve, The Source, etc.) you may encounter certain problems in making a viable connection. Some of the most common problems are: wrong Baud Rate; the other system will not lock carrier (probably only supports 300 baud, switch to 300 and try again); wrong Data Format; you have locked carrier but there is garbage coming in on the screen (typically change from NO parity to EVEN).

For good information and for a wealth of telecommunications resources, look into these books: The OMNI series on Telecommunications (including the Online Database Directory), Technical Aspects of Data Communications (Digital Press), The RS232 Solution (Sybex Books) and for a look at how some of these services work, read The Complete Handbook of Personal Computer Communications (St. Martin's Press).

I have had the opportunity to briefly use three new telecommunications packages available now for the Macintosh:

inTouch by Software Masters, 3330 Hillcroft 'BB', Houston, Tx. 77057 713-266-5771, \$149.95. (128 or 512K)

MacLine by TouchStone Software, 909 Electric Ave, Seal Beach, Ca. 90740

213-598-7746, \$145.00. (128 or 512K)

SmartCom II by Hayes Microcomputer Products, 5923 Peachtree Industrial Blvd, Norcross, Ga. 30092, 404-449-8791, \$149.00. (128 or 512K)

Each of these packages has unique capabilities, and all have more functionality than MacTerminal from Apple. All these programs allow you to configure the communications protocol settings that you desire.

"inTouch" is a very well written telecomm program, capable of many styles of Terminal Emulation (TTY, Beehive, Televideo 925, IBM 3101, ADDS Viewpoint, DEC vt100 and even VIDTEX graphics). It supports text file and binary file transfers including Xmodem & Crosstalk protocols and a special "inTouch" protocol for Mac to Mac file transfers. "inTouch" has 8 function (macro) keys and also

programmable keys for "username" & "password". There are also keys to represent the DELETE, ESCAPE & BREAK keys found on terminals. These keys are all softkeys addressed with a click of the mouse.

There is an onscreen clock that also functions as an online timer; a phone number setting (with an auto redial function if the number is busy); a screen scroll buffer (40 lines on a 128K, 400 on a 512K); ability to print incoming data in real time; save text to disk; manipulate incoming and outgoing CR&LF information; utilization of a MiniFinder for selecting files for transfer; ability to view text files; support of auto-answer mode; an auto-execute function, and "inTouch" calls Communications Command Language, which is a very flexible batch command system that allows unattended telecommunications at predetermined times.

You can write CCL files that will dial up Compuserve and logon for you, change to the appropriate area of the system, extract the information you want and log you off, or you could write a CCL file that would call a number of other Macs running 'inTouch' and transfer specific files to each machine at specific times. All in all a very nicely done, nicely documented package.

[An added feature is IBM 3101 terminal emulation. Think you need Apple's \$1000 box to talk to IBM networks? Not so. Since IBM started making an ASCII terminal (the 3101), they have had to support it on their networks. The end result is that most IBM networks have IBM software that provides full IBM 3278 full screen editing emulation for IBM 3101 Ascii terminals on the network. You dial up and log on as a 3101 character mode terminal, then switch to block mode transmission, and the IBM software emulates 3278 protocol for you. Hence, all that is needed to get full 3278 emulation is a software package capable of emulating an IBM 3101 ASCII terminal sufficiently to "fool" the IBM software into providing 3278 emulation rather than "dumb terminal" emulation. We have tested the "inTouch" program and while it does communicate with IBM systems as a 3101 terminal, it presently lacks the necessary block mode transmission to invoke the 3278 support from IBM's side. However, we have talked to the company and they are very enthusiastic about attempting to add whatever is necessary to get the 3278 support. - Ed.]

MacLine is a rather unique telecomm product. While it performs admirably as a stand-alone package, how it fits into a family of software written for many operating systems is unique. TouchStone presently markets PCworks for the IBM-PC (and most clones), MacLine for the Macintosh and UniHost for a half dozen UNIX machines (I hear they are working on a DEC Vax VMS version also). These packages address the incompatibility of the different operating systems. In different ways, they all use a similar user interface to maintain familiarity among the systems. The interface is user-definable and menu driven and divided into four basic areas, Terminal Communications, Request Network Services, Provide Network Services and Start Programs.

Terminal Emulation allows for direct and modem connections, emulation of TTY, ANSI and DEC VT 52/100 terminals; this area of the main menu is actually a minifinder-like scroll window that allows you to scroll through as many predefined terminal hookups (names, phone numbers, comm parameters and emulation) as you have disk space for.

The Start Programs area of the main menu is once again a scroll window (as are all of the windows on MacLine) and it allows you define a macro to start local Macintosh programs and remote Unix programs (dialing, logging you in, changing to the appropriate directory and executing the application) all from inside of MacLine. Once the application is finished you will come back to the main menu of MacLine once again (the usefulness of this feature is obvious).

The Provide Network Services window allows you to define multiple set-ups expressly for unattended file transfers. Each definition allows you to assign user account names and passwords, with read/ write and overwrite permissions also assignable; and finally there is a provision for mail services within this area. This area turns your Mac into a secured Host.

The Request Network Services window is where it all comes together. In this area you can define the system you are calling, the logon, password, default directory, spooler label, terminal emulation and modem number (if you are using a dial-up rather than direct). Within this area (which is also menu driven) there is a built-in editor, so you can create a letter and mail it, load a letter from disk and mail it (there is an interface to UNIX mail and also a Mail system designed into each TouchStone package); send a file to the print spooler; print one locally; send a file to the Host system (text or binary) and convert it for that format or archive it there for future retrieval. You get a file from the Host in exactly the same fashion as sending it (conversion or archiving).

There is a lot to this program, but there are a few things that hamper it (which I am told are being addressed in a new version being prepared for release). It does not support Xmodem (TouchStone uses a proprietary protocol which is very stable but it's not Xmodem), and you cannot capture a file to disk in the Terminal Emulator (although you can Send from disk). MacLine is a unique program with good documentation.

SmartCom II is probably the most "friendly" package for telecomm I have ever seen, and its use of icons is very well laid out and quite functional.

The program supports terminal emulation of TTY, DEC VT 52/100; autodial & redial if busy; a batch command system called "Autopilot" for unattended functions like time dependent logons, file transfers, etc.; file capture (select a piece of text already in the buffer and then save to disk or capture all incoming data to disk in real time); Xmodem & Smartcom protocol transfers; local print (select a piece of text already in the buffer and then print or print all incoming data in real time); text file transfer from disk; auto-answer; smooth scroll option; and one interesting and completely unique feature: INTERACTIVE GRAPHICS (requires 512K). Two

people running Smartcom on 512K Macs can create and/or modify drawings on-line. You can load a MacPaint file from the Scrapbook and utilize that on what Smartcom calls their "Canvas" or start from scratch. Hayes has even thoughtfully provided a chess board with all the pieces to allow you to play a real interactive VISUAL chess game through the modem! The implications and applications stagger the mind. Smartcom is a very nicely done package with excellent documentation.

All of these programs worked well for me, some had special features that were unique to themselves. All of them were miles away from MacTerminal and its quirks. I would not hesitate to recommend that anyone evaluate them further to see if they fit their personal needs.

Happy Communicatin'!

Bruce

(logoff)

Asynchronous Telecommunications Glossary

Asynchronous: The transmission of characters, serially (one bit at a time), with start and stop bits separating each byte of data. The start and stop bits "frame" the data.

Subminiature "D" - connector: Called a "D" connector because the chamfered corners give the appearance of the letter D, this connector format is the standard used in communications hardware. The db-25 and db-9 connectors have 25 and 9 pins respectively.

RS232C: A low speed serial data communications "standard" defined by the EIA (Electronics Industry Association). The standard defines a number of parameters such as voltage levels, loading characteristics and timing relationships. RS232C is capable of baud rates up to 19.2K (cable runs at that speed are limited to 50 feet).

"db25" Pin Assignments

Pin 1. Chassis Ground (CG) - Also known as Frame Ground, this connection insures that the chassis of both devices are at the same electrical potential, to prevent electrical shock and other electrical faults. This connection is optional.

Pin 2. Transmit Data (TD) - This circuit is the path whereby serial data is sent from DTE to DCE. This circuit must be present if data is to travel in that direction at any time.

Pin 3. Receive Data (RD) - This circuit is the path whereby serial data is sent from DCE to DTE, and also must be present if data is to travel in that direction at any time.

Pin 4. Request To Send (RTS) - A signal from a DTE to a DCE to prepare the DCE for data transmission.

Pin 5. Clear To Send (CTS) - A signal from a DCE to a DTE to a DTE that stays false until the DCE makes it true, indicating that all circuits are ready to transfer data.

Pin 6. Data Set Ready (DSR) - A signal from a DCE to a DTE indicating that the DCE has established a connection.

Pin 7. Signal Ground (SG) - This circuit is the ground to which all other voltages are referenced. It must be present in any RS232C interface.

Pin 8. Data Carrier Detect (DCD) - A signal from a DCE to a DTE indicating that it has an incoming carrier and that a communication connection has been established.

Pin 20. Data Terminal Ready (DTR) - A signal from a DCE to a DTE indicating readiness to transmit or receive data.

Pin 22. Ring Indicator (RI) - An optional signal from a DCE to a DTE that indicates the arrival of a call.

RS422: An expanded and higher speed version of RS232C. The Mac's serial ports are internally clocked at 230Kbaud and support externally clocked speeds of 920Kbaud. RS422 has better noise rejection and can support a much longer cable run than RS232C. As the signal format of the Mac SCC (for telecommunications) is an emulation of RS232C functions, we will continue to refer to those functions and not those of RS422.

"db9" Pin Assignments

Pin 1. (CG), Pin 3. (SG), Pin 5. (TD), Pin 6. (filtered +12 Volts), Pin 7. (CD), Pin 9. (RD).

ASCII: American Standard Code for Information Interchange; assigns a number from 0-255 to each text (and IBM PC character graphics) character and control function character. Many of you deal with Big Blue (IBM) everyday on the PC level. Keeping in mind the growing number of IBM PC-based Bulletin Board Systems and how much they utilize the special PC character graphics set, you should be aware of the following information: the special IBM PC ASCII characters have ASCII values above 127 (see Appendix G of your IBM PC BASIC manual). These characters cannot be transmitted or received when the system is using 7 data bits; the high-bit is assumed to be a parity bit and is stripped from the data in the transfer process, leaving a character with an ASCII value 128 less than the one contained in the original file.

Baudot: 5 bit code used w/ Telex and DeafNet (typically 45-110 baud).

Baud-Rate: A unit of measurement of serial communication speed, in bits per second (bps), for phone lines it is typically 300 or 1200 baud; for hardwired communications (printers, terminals) it's normally 9600 baud.

Binary (base 2): A number system with only two digits, 1 & 0 (On & Off, Hi & Lo, True & False) with each digit position moving from right to left representing a successive power of two (ie; decimal 1= binary 1, decimal 2= binary 10, decimal 4= binary 100, decimal 8= binary 1000...).

Bits & Bytes: A binary digit, the most fundamental of

digital information, it represents one of two values or "states", 0 or 1 (On or Off). Eight bits comprise a byte and can represent a value between 0 and 255.

Break-out Box: An easily configurable test device that enables discrete viewing and control of most RS232 lines.

Buffer: A section of memory that acts as a reservoir between one device and another (information coming from a modem or going to a printer are typical applications for a buffer), the buffer acts as reservoir if the data is being sent faster than the receiving device can process it.

Capture: Opening a "capture" buffer and reading messages, e-mail, bulletins, etc into that buffer and then writing it to the disk when finished (you can also capture directly to disk).

Carrier: An audio signal that "carries" the data across the phone lines. The pitch that you hear when you connect and that your modem answers with (300 baud answers with a pure tone, 1200 answers with shaped noise). The modem "MODulates" and "DEModulates" the digital data from digital bits to analog audio frequencies, the carrier is a constant frequency signal that maintains the connection between data being transmitted. The Bell standard frequencies for Originate & Answer are most commonly used in the microcomputer world (hence the description Bell 103 or Bell 212 for modem standards). The CCITT carrier standard is set by the International Telegraph & Telephone Consultative Committee and is far less common.

Carrier frequency definitions for Bell 103 :

Signal	State	Originate	Answer
logic 0	space	1270	2225 (cps)
carrier	-----	1170	2125
logic 1	mark	1070	2025

Character (or Data) Format: The combined information of the Data Bits, the Parity Bits and the Stop Bits.

Chat: Two-way conversation via keyboard, normally between the SYSOP (System Operator) and yourself. Chat courtesy protocol: two carriage returns when you are finished typing (keeps the conversation and the screen clean). "GA", for "Go Ahead" is also frequently used.

Control Characters (codes): A non-visible ASCII character that performs specific functions (like screen formatting, line feeds (^J) <LF>, carriage returns (^M) <CR>, form feeds (^L) <FF>, end of file markers (^Z) <EOF>, nulls (^A), escape (^D), continue, abort, etc). The "^" character is commonly used to represent a CONTROL character.

CRC: Cyclical Redundancy Check, used for 16 bit error correction schemes, similar to checksum but more accurate.

Data Bits: Several bits of information, normally 7 or 8, which represent ASCII characters.

DCE: Data Communications Equipment (transmit on 3, receive on 2): the modem.

Download: Receiving a pre-written text-file or binary-file from the host system.

DTE: Data Terminal Equipment (transmit on 2, receive on 3): the computer.

Duplex: Convention to determine directionality of the data flow, ie: Full Duplex allows simultaneous transmission from both the Host and Guest (like the telephone system); Half Duplex allows only singular transmission in either direction (like a set of walkie talkies).

EBCDIC: Mainframe equivalent of the ASCII standard (Expanded Binary Coded Decimal Interchange Code).

Echo: Used to verify character transmission to the opposing party, if you type "hello" and see "hheelllloo" on your screen, turn echo off (or change Half-Duplex to Full).

Filters: Some communications software allows the user to define characters that they want screened (or filtered) from incoming and outgoing data (ie; catching a control L so the screen doesn't clear).

Flow Control: Software Handshaking, ie; Xon/ Xoff protocol (control Q & control S) and Etx/ Ack protocol (control D & control F)

FSK, PSK: The techniques used by Bell 103 and Bell 212 modems, Frequency Shift Keyed and Phase Shift Keyed.

Handshaking: Hardware or Software handshaking is used to keep a steady flow of information between devices without loss. Hardware handshaking uses RS232C control signals from the sending and receiving devices to tell when the communications buffer is getting full and to temporarily halt the flow of data from the sending device until the receiving buffer is sufficiently clear to handle incoming data again; Software handshaking is designed around intelligent software that recognizes the state of the buffer and uses control characters to communicate that information and to control the flow of data.

Hex: Hexidecimal Code (base 16), a numbering system that uses 16 digits; usually these are represented by the ten decimal digits, 0 through 9, plus the letters "A" through "F" ("A" representing decimal 10, "F" representing decimal 15, etc.). Each hexadecimal digit can represent a string of four binary digits.

Macros: A string of ASCII assigned to a predefined key (ie; your system account name, password, or anything you please).

Native File Format: Operating Systems store information in different ways, almost all of them incompatible with one another. "Flat" ascii data files (non-formatted, ie: MacWrite in "text only" mode) can be moved from one system to another, but not without a modicum of difficulty. The major stumbling blocks are the Carriage Returns, Line Feeds and End Of File markers. Macintosh O.S. terminates each line of text with a <CR>, the last line of text in the document is NOT terminated; IBM PC-DOS terminated each line of text with a <CR><LF> pair and terminates the last line of text in the document with a <CR><LF> pair and then an <EOF> marker at the very end. UNIX Systems terminate each line of text with a <LF>, and the last line of text in the document is terminated with a <LF> on its own line. Hence the confusion rampant in the telecommunications world: you cannot simply grab a file from one machine and use it right away in a foreign O.S.. It must be converted to conform to the standards of the O.S. that you intend to read/manipulate it with.

Parity: The parity bit is set to 1 or 0 to modify the logical sum of the bits in a transmitted byte to be either even or odd (ie; to achieve even parity, if the sum of the bits in a byte is already even the parity bit is left as 0 if the sum is odd, setting the parity bit to 1 will make the byte even). Parity is used for error detection.

Protocol: Communications Settings (ie; Data Format, Baud Rate, File Transfer Method).

Start Bit: A start bit defines the arrival of a character on an RS232C communications link.

Stop Bits: One or two stop bits separate adjacent characters on an RS232C communications link.

Terminal Emulation: Some communications software allows your computer to emulate the terminals that are commonly used on multi-user systems so you may connect to such a system either directly or remotely (via modem) and have all the screen formatting and other terminal functions the same as they would be on the actual terminal, with the added ability to use the intelligence of your computer (ie; capturing data to disk, uploading data from disk).

Upload: Sending a pre-written text-file or binary-file to the host system.

XMODEM: Protocol Transfer via fixed length blocks of data, using Enquiry (ENQ) along with Acknowledge (ACK) and Negative Acknowledge (NAK); Start Of Header (SOH) and End Of Transmission (EOT). Blocks of data are sent across, checked for the proper header and block checksum. If any errors are detected the data is re-transmitted until the block is verified, or after multiple retries it gives a "time-out" and aborts the transfer (this is common on Sprint and MCI lines). Xmodem is a public domain protocol created by Ward Christensen that has gained widespread acceptance as a file

transfer standard. Other protocols that you will possibly encounter are Kermit (also public domain, from Columbia University), Smartcom (from Hayes) and MNP (from MicroCom).

Cable recommendation for Mac to Smartmodem

db9	to	db25
Pins 3 & 8	to	Pin 7
Pin 5	to	Pin 2
Pin 6	to	Pin 20
Pin 7	to	Pin 8
Pin 9	to	Pin 3

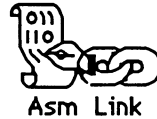
Areas Of Future Interest

Synchronous communications and protocols...BCP's (byte control protocols); DDCMP (D.E.C. digital data communications message protocol), Bisync (IBM mainfram standard) and BOP's (bit oriented protocols); IBM's SNA (System Network Architecture) which uses SDLC (synchronous data link control) a version of the internationally used HDLC (high-level data link control); Packet switched networks...



Programmer's Forum

PrLink Source for MacAsm



Paul F. Snively
MacTutor Contributing Editor
MacTutor Vol. 1 No. 9

Well, if you have been following MacTutor for any length of time, you are probably aware that we are dedicated to the proposition that the Mac can be programmed with any of a number of languages available for the Mac, on the Mac.

What does it mean to program the Mac, though? Much has been made of the "standard user interface guidelines," which is a fancy way of saying that Apple Computer, Inc. thinks that all Mac applications should bear at least a passing resemblance to each other. From the user's point of view, this is a Godsend.

If you have read Inside Macintosh you are aware that there are at least three things that Apple says should be consistent from application to application: the apple menu, the file menu, and the edit menu.

Several people have gone into detail as to how to create a full-featured apple menu and edit menu for Mac applications. More recently, the "Open..." option (i.e. the standard file interface) has gotten a close look. Until now, however, no one has really looked at the obvious remaining element in a Mac application, namely, the "Print..." option.

I'm going to dodge the menu setup; I detest reinventing the wheel. If you are unsure as to how to create a "Print..." menu option which prints the current document associated with your application, I suggest that you get some MacTutor back issues and start reading. I am only going to concern myself here with the actual printing of text and graphics from within an assembly language application.

PrLink Source Missing

If you are using the release version of MDS, you are probably wondering what the big deal is. After all, MDS includes a file called PrLink.Rel which can simply be linked into your application and provides support for all of the printing manager routines. This is entirely true. If you are using MDS, don't bother to continue reading unless the theory behind the printing manager interface interests you, because the code is written for MacASM owners, who weren't blessed with a canned interface to the printing manager. [Note that Apple did not supply the source code to PrLink.Rel. How come? -Ed.]

Allow me to editorialize for a moment here. (Contributing editors can do that, can't they?) There is probably no area of the toolbox (if it can properly be called "part of the toolbox") that is so shrouded in mystery as the printing manager. The only program that I own that actually works through the printing manager is MacWrite (well, perhaps MacPaint does too, although I think it works at a

lower level than MacWrite does). Of all of the development tools that I own, NOT ONE of them uses the printing manager -they simply open the serial port and start sending ASCII codes out. This is hardly the most device-independent way of doing things; as David Smith has noticed, that technique tends not to work with the LaserWriter, for example. So it is important to work through the printing manager for reasons of compatibility, as well as to be able to use the font manager and Quickdraw to create graphics on the printer.

The primary reason that the printing manager is so neglected is that, unlike most areas of the toolbox, there is virtually no correlation between how the printing manager is accessed in PASCAL and how it is accessed in assembler (once again, unless you are using MDS - but even at that PrLink.Rel won't win any awards for being well-documented). In PASCAL, you simply use the appropriate functions and/or procedures, giving the proper parameters - no big deal. In assembly, though, you must deal with the printing resource file, overlays, dialog boxes, and so on - right down at the nuts and bolts level. The code that follows is my attempt to provide much the same easy access to the printing manager routines as PrLink.Rel does for MDS users.

Code Resources

If you have been following the whole discussion about resources, you have probably become aware that they are found (for the most part) in two places: within the application's file and within the system resource file (called, aptly, "System"). What you may or may not be aware of is that your application can also open other resource files to draw needed resources from. Opening resource files creates entries in a linked list: the most recently opened resource file is the first one searched, with the application itself being second to last, and the system resource file last.

The printing manager routines exist as resources in the printing resource file. Since resources can be executable code, Apple decided to use that approach (probably in order to allow for device independence as far as printers go - it'd be mighty hard to support a laser printer if the printing manager routines were in ROM). Interfacing to the printing manager consist of loading these resources as necessary and executing the code within them.

My Own PrLink Source

The "Print.Asm" code takes care of all of the dirty work for us by giving us the same routines as PASCAL users have. These routines can be described as follows:

The PrOpen routine opens the printing resource file and also opens the printer driver for us. We can then use the driver directly or through the high-level printing manager routines.

The PrStdDialog function is most commonly associated with the "Page Setup..." menu option. It allows the user to describe the page to the computer.

The PrJobDialog function is the one that is actually invoked upon choosing "Print..." and consists of the usual choices regarding pages to print, type of paper to use, and quality of printing.

The PrintDefault procedure simply fills in the PrintRec with the default values associated with that printer. This is useful for preventing weird values from popping up in PrintRec and for making it possible to set up a commonly used set of values (the default changes every time you answer the dialogs).

PrValidate makes sure that the PrintRec you pass to it is valid as far as the printing manager is concerned. This is handy if your application expects to find the PrintRec for a document within the document itself (yes, you can do that). Simply read the PrintRec from the document and do a PrValidate on it. PrValidate will fail (returning a boolean "false") if the PrintRec is not compatible with the current printing manager software.

PrJobMerge is useful if your application allows for printing of multiple documents in succession (in other words, if it allows the user to select several document icons from the Finder and use the "Print..." option in the Finder's file menu - technically, any application that allows printing should allow this). PrJobMerge copies the results of the dialogs into all of the PrintRecs for the documents that were passed to it, so that the user's selections will be used for all of the documents.

Asm Routines Mimic Pascal Interface

PrOpenDoc is the procedure that is actually responsible for preparing the printing manager to accept printer instructions. What this routine actually does depends upon whether the user selected draft printing or not. In draft, all operations are sent to the printer without further ado. With standard or high quality printing, the pages are sent to disk first, and then are printed using the PrPicFile procedure.

PrOpenPage starts a new page. This is particularly important in non-draft printing, since what essentially happens is that Quickdraw draws the "page" as a picture, except that it stores it on disk instead of putting it in a bitmap in RAM. PrOpenPage is then the printing equivalent of the OpenPicture toolbox trap.

PrClosePage is the printing equivalent of ClosePicture. It lets Quickdraw know that this picture (page) is done.

PrCloseDoc lets the printing manager know that the document is finished. The primary purpose of this seems to be for non-draft printing. It removes the special routing of Quickdraw traps and allows Quickdraw to work with bitmaps again instead of the printer.

PrPicFile is the procedure that prints non-draft quality documents. It allows for background processing (which

usually consists of popping up a dialog box pointing out that printing is taking place and can be cancelled by holding down the command and period keys, and, upon return from the printing manager, removing the dialog).

All of these assembly language routines are intended to look exactly like their PASCAL counterparts in setup. The rule is that whenever a PASCAL function or procedure is described, the assembly equivalent takes its parameters in the same order as the PASCAL routine (in other words, the first parameter to a PASCAL routine is the first thing pushed for the assembly routine). This code obeys that particular rule, as you will see upon examining the sample program "Print-Examp.Asm." [The MacASM code that links you, the programmer, to the printing manager begins on the next page].

This printing manager interface took a lot of reading about memory management, device driver management, and the printing manager to work out. You may notice that the only low level operations performed are opening and closing the printer driver (and some of you may be wondering why I tossed in the printer driver close). I have found no use for the low level printer driver routines, so I didn't include them. Feel free to add them if you wish.

I added the printer driver close because I'm kind of a neatnik when it comes to what I leave lying around in my code. If I need to open the driver, then I prefer to close it when I am through with it. Since PrClose does not close the driver (it only closes the printer resource file, in accordance with IM), I added DCLOSE out of my own sense of propriety.

Apple's Printing Example

Now that you can see how to get from the description of the PASCAL routines to the assembly language equivalent, how do you use these routines? Answering that question is the purpose of the program "PrintExamp.Asm."

"PrintExamp.Asm" is simply the MacASM version of the "TestPrint.Asm" source code that was included with MDS. THIS CODE IS ORIGINAL ONLY IN THAT I REWROTE IT FOR MACASM!

Print Example In Assembly

The PrintExamp program teaches us several things about the interface to the printing manager. First, it succeeds at working exactly like PASCAL routines (the first parameters listed are the first ones on the stack). Second, it is self-contained (you need not define any globals or what have you for the interface per se). Third, it works.

Most importantly, PrintExamp shows how to determine how much Quickdraw material will fit on a page. Examine lines 590-690 of PrintExamp closely. This code calculates the pertinent info about the size of the page by using the GetFontInfo toolbox trap (plus a little simple mathematics).

00010 ;SAVE "Print.Asm"

00020 PrOpen	BSR	DOPEN	;Open the printer driver
00030	BNE.S	NOERR	;If an error occurred, exit
00040	MOVEQ	#\$00,D1	;Flag for open
00050	BRA.S	GETNAME	;Get printer resource filename
00060 PrClose	MOVEQ	#\$01,D1	;Flag for close
00070 GETNAME	SUBQ	#\$04,SP	;Make room for result
00080	MOVE.L	#\$53545220,-(SP)	;Resource type "STR "
00090	MOVE.W	#\$E000,-(SP)	;ID # \$E000
00100	TBX	GetResource	;Self-explanatory
00110	MOVE.L	(SP)+,D0	;Get handle
00120	BEQ.S	NILHAND	;Error if handle = NIL
00130	MOVEA.L	D0,A1	;Put handle in A1
00140	SUBQ	#\$02,SP	;Make space for result
00150	BSET	#\$0007,(A1)	;Lock the master pointer
00160	MOVE.L	(A1),-(SP)	;Move master pointer to stack
00170	TBX	OpenResFile	;Open printer resource file
00180	MOVE.W	(SP)+,D0	;Get result
00190	BCLR	#\$0007,(A1)	;Unlock master pointer
00200	BSR.S	ERRCHK	;Was there a problem?
00210	TST.W	D1	;OPEN or CLOSE call?
00220	BEQ.S	NOERR	;If OPEN, we're done
00230	MOVE.W	D0,-(SP)	;Else drop result back on stack
00240	TBX	CloseResFile	;And close the printer res file
00250 ERRCHK	MOVE.W	\$0A60,\$0944	;Move resource err to PrErr
00260	BEQ.S	NOERR	;If there was no problem, return
00270	ADDQ.W	#\$04,SP	;Else drop return address
00280 NOERR	RTS		;Return to caller
00290 NILHAND	MOVE.W	\$\$FF40,\$0944	;Out of memory error
00300	RTS		;Return to caller
00310 PrintDefault	MOVE.W	#\$8000,D1	;Offset 0
00320	BRA.S	OVL4	;Execute overlay code
00330 PrStIDialog	MOVE.W	#\$8004,D1	;Offset 4
00340	BRA.S	OVL4	;Execute overlay code
00350 PrJobDialog	MOVE.W	#\$8008,D1	;Offset 8
00360	BRA.S	OVL4	;Execute overlay code
00370 PrValidate	MOVE.W	#\$8018,D1	;Offset \$18
00380	BRA.S	OVL4	;Execute overlay code
00390 PrJobMerge	MOVE.W	#\$801C,D1	;Offset \$1C
00400	BRA.S	OVL4	;Execute overlay code
00410 PrOpenDoc	MOVEA.L	\$000C(SP),A0	;Get handle to print record
00420	MOVEA.L	(A0),A0	;Dereference handle
00430	MOVEQ	#\$03,D0	;Mask for all possible ptypes
00440	AND.B	\$0044(A0),D0	;Which bit in record is set?
00450	MOVEQ	\$\$FFFFFFFC,D1	;All bits but lowest 2
00460	AND.B	D1,\$0946	;Mask off lowest 2 bits
00470	OR.B	D0,\$0946	;Set correct ptype bit
00480	MOVEQ	#\$00,D1	;Offset 0
00490	BRA.S	OVL3	;Adjust for proper overlay

00500 PrCloseDoc 00510	MOVE.W BRA.S	#\$8004,D1 OVL3	;Offset 4 ;Adjust for proper overlay
00520 PrOpenPage 00530	MOVEQ BRA.S	#\$08,D1 OVL3	;Offset 8 ;Adjust for proper overlay
00540 PrClosePage 00550	MOVEQ BRA.S	#\$0C,D1 OVL3	;Offset \$C ;Adjust for proper overlay
00560 PrPicFile 00570 00580	MOVEQ MOVEQ BRA.S	#\$00,D1 #\$05,D0 EXECOVL	;Offset 0 ;Overlay 5 ;Execute overlay code
00590 OVL4 00600	MOVEQ BRA.S	#\$04,D0 EXECOVL	;Overlay 4 ;Execute overlay code
00610 OVL3 00620	MOVEQ AND.B	#\$03,D0 \$0946,D0	;Mask for overlay bit ;Mask off all but correct bit
00630 EXECOVL 00640 00650 00660 00670 00680 00690 00700 00710 00720 00730 00740 00750 00760 00770 00780 00790 00800 00810 00820 00830 00840 00850 00860 00870 00880 NOUNLOCK 00890 00900 00910	LEA MOVEM.L MOVE.L MOVEA.L LEA MOVE.L SUBQ.L MOVE.L TBX MOVE.L BEQ.S MOVEA.L BSET MOVEA.L MOVEQ MOVE.B ADDA.L MOVE.L CLR.B MOVEA.L JSR TST.W BPL.S BCLR MOVEA.L LEA MOVEM.L JMP	SAVEREGS(PC),A0 D4/A3/A4,(A0) D1,D4 (SP)+,A3 SAVESTK(PC),A0 SP,(A0) #\$04,SP #\$50444546,-(SP) D0,-(SP) GetResource (SP)+,D0 OVLERR D0,A4 #\$0007,(A4) (A4),A0 #\$00,D0 D4,D0 D0,A0 A0,-(SP) (SP) (SP)+,A0 (A0) D4 NOUNLOCK #\$0007,(A4) A3,A1 SAVEREGS(PC),A0 (A0),D4/A3/A4 (A1)	;Register save area ;Save pertinent registers ;Copy offset ;Get return address ;Point to stack save area ;Save current stack address ;Make room for result ;"PDEF" type resource ;Which overlay? ;Get that overlay ;Handle to overlay ;If no handle, overlay error ;Put handle in address reg 4 ;Lock master pointer ;Get master pointer in A0 ;Zero out D0 ;Move offset to D0 ;Add offset to master pointer ;Put result on stack ;Zero out first byte's garbage ;Get real address in A0 ;Execute overlay code ;Check out D4 ;Go if no unlock needed ;Else unlock master pointer ;Return address to A1 ;Point to register save area ;Get regs contents back ;And return to caller
00920 OVLERR 00930 00940	MOVE.W MOVEA.L BRA.S	#\$FF40,\$0944 SAVESTK(PC),SP NOUNLOCK	;Put error code in PrintVars ;Get pristine stack back ;Back to calling program
00950 SAVEREGS 00960 SAVESTK	DEFS DEFS	12 4	;12 bytes for registers ;4 bytes for the stack pointer
00970 DOPEN 00980 CLEAR 00990 01000	MOVEQ CLR.W DBRA LEA	#\$18,D0 -(SP) D0,CLEAR DVRNAME(PC),A0	;Clear \$19 words on stack ;Pointer to driver name

01010	MOVE.L	A0,\$0012(SP)	;Place in data structure
01020	MOVEA.L	SP,A0	;Point to beginning of data
01030	OST	Open	;Open the driver
01040 CLEANUP	ADDA.W	#\$32,SP	;Remove data from stack
01050	MOVE.W	D0,\$0944	;Save result in PrintVars
01060	RTS		;And return to caller
01070 DCLOSE	SUBA.W	#\$32,SP	;Make room for data structure
01080	MOVEA.L	SP,A0	;Point to data structure
01090	MOVE.W	#\$FFFD,\$0018(A0)	;Move reference number to data
01100	OST	Close	;Close the driver
01110	BRA.S	CLEANUP	;Clean up stack and return
01120 DVRNAME	STR	".Print"	;Name of printer driver
01130	ADJUST		

If you are dealing with graphics objects other than fonts, the pertinent information to remember is that there are 72 dots per inch both on the Macintosh screen and on the printer (which is how the coordinates for a 5" by 5" oval become 0,0,5*72,5*72).

The major data structure associated with the printing manager routines is the PrintRec. For the most part, all you have to worry about is allocating it and keeping track of the handle (see PrintExamp.Asm and IM for more information).

Your application's file menu should ideally have both "Page Setup..." and "Print..." options. "Page Setup..." is associated with the PrStlDialog. "Print..." is associated with the PrJobDialog, and then with the routines necessary to actually do the printing.

Your application should also check the Finder info block upon startup to see if it needs to print more than one document. If it does, it should use PrJobMerge to apply the results of the dialogs to all documents in the batch.

Well, that wraps up my coverage of the printing manager! If you have any further questions, please drop me a line c/o MacTutor. Enjoy!



MacAsm to MDS Syntax

MDS users may find the non-standard syntax of the Mac Asm source code a bit confusing. Here is a short table of some of the differences found in the code in this article.

ADJUST: Adjust boundary (.ALIGN)
 ASC: String definition (DC)
 DATA: Data definition (DC)
 DEFS: Define Storage (DCB)
 ENDM: End macro def. (l, .ENDM)
 ENDR: End resource def.
 RSRC: Start resource def.
 GLOBAL: Storage (A5) as in (DS)
 TBX: Trap Macro (_Open)
 OST: OS Macro (_Open)
 STR: String with length (DC)


```

00010 ;SAVE "PrintExamp.Asm"
00020     LIST    OFF
00030 *-----
00040 ; Macintosh printing manager example program
00050 ; Copyright (c) 1985 MacTutor
00060 *-----
00070 ; Example source code translated from MDS format by Paul F. Snively
00080 *-----
00090 ; "Print.Asm" include file (interface to printing manager) written by
00100 ; Paul F. Snively
00110 *-----
00120 charCount      EQU      120           ;Characters to print per line
00130 countAndLen    EQU      charCount+2   ;charCount and length byte
00140 monaco         EQU      4           ;Monaco is font # 4
00150 ascent         EQU      0           ;Offset into font info record
00160 descent        EQU      2           ;ditto
00170 leading        EQU      6           ;ditto
00180 prInfo         EQU      2           ;Offset into printer info rec
00190 rPage         EQU      6           ;ditto
00200 bottom         EQU      4           ;Offset into rect structure
00210 iPrintSize     EQU      120          ;Print Record size
00220 iPrStatSize    EQU      26          ;Size of printer status record 00230 *---
-----
00240                INCLUDE    "Library.Asm"
00250 *-----
00260                GLOBAL    L+iPrStatSize,$CE
00270                DEFV      L,hPrintRec
00280                DEFV      iPrStatSize,prStatus
00290                ENDG
00300 *-----
00310                TFILE      "Buffer.Bin"           ;Target file for assembly
00320                RFILE      "PrintExamp",APPL,PRTX,$2000 ;Resource file
00330 *-----
00340                SEG      1,52
00350 *-----
00360 Start          BSR      InitManagers      ;Initialize managers
00370                BSR      PrOpen           ;Open print manager
00380                BRA      EventLoop        ;Go start event loop
00390 *-----
00400 InitManagers  PEA      -4(A5)           ;Standard init sequence
00410                TBX      InitGraf         ;Init Quickdraw
00420                TBX      InitFonts        ;Init Font manager
00430                MOVE.L    #$0000FFFF,D0   ;Flush all events
00440                TBX      FlushEvents
00450                TBX      InitWindows       ;Init Window manager
00460                TBX      InitMenus        ;Init Menu manager
00470                CLR.L     -(SP)           ;No restart procedure
00480                TBX      InitDialogs      ;Init Dialog manager
00490                TBX      TEInit          ;Init Text Edit
00500                TBX      InitCursor       ;Turn on arrow cursor
00510                RTS
00520 *-----
00530 MyDrawPage    MOVEM.L D3-D6/A3-A4,-(SP) ;Save registers
00540                PEA      tRect(PC)        ;Push tRect
00550                TBX      FrameOval        ;Draw a 5" by 5" oval
00560                MOVE     #monaco,-(SP)    ;Push monaco font #
00570                TBX      TextFont        ;Set font (default size)

00580 ;Get font info to determine line spacing

00590                LINK      A6,#-8         ;Make room for fontInfo

```

```

00600          MOVE.L    SP,A4          ;A4 points to fontInfo
00610          MOVE.L    A4,-(SP)      ;Push pointer to fontInfo
00620          TBX        GetFontInfo  ;Get the fontInfo record
00630          MOVE      ascent(A4),D4 ;Calculate line height
00640          ADD        descent(A4),D4
00650          ADD        leading(A4),D4 ;D4 has line height
00660          MOVE.L     hPrintRec(A5),A0 ;Point to print record
00670          MOVE.L     (A0),A0       ;dereference handle
00680          MOVE      prInfo+rPage+bottom(A0),D6 ;Get page bottom coord.
00690          SUB        descent(A4),D6 ;Adjust for font descent
00700 *-----
00710 ;Print a page of characters.
00720 ;
00730 ;A3 points to the print string
00740 ;
00750 ;D3 has current line position
00760 ;D4 has vertical distance between lines
00770 ;D5 has current line number
00780 ;D6 has bottom (vertical) coordinate of page
00790 ;
00800          MOVE      #1,D5          ;Set initial line no.
00810          MOVE      D4,D3          ;Set initial line position
00820          LINK      A6,#-countAndLen ;Space for char string
00830          MOVE.L     SP,A3          ;A3 points to string
00840          MOVE.B     #charCount,(A3) ;Set length byte

00850 ;Fill print string with characters.

00860          MOVE      #$20,D0        ;ASCII value for space
00870          MOVE      #charCount-1,D1 ;Count-1
00880          MOVE.L     A3,A0          ;Point to string
00890          ADDQ       #1,A0          ;Bump past length byte
00900 .1      MOVE.B     D0,(A0)+      ;Fill in string
00910          ADDQ       #1,D0          ;Next char
00920          DBRA      D1,.1          ;Loop until done
00930 Ploop   MOVE      #0,-(SP)       ;MoveTo start of line
00940          MOVE      D3,-(SP)
00950          TBX        MoveTo
00960          MOVE.L     A3,-(SP)       ;Draw string
00970          TBX        DrawString
00980          MOVE      #charCount-1,D1 ;Count-1
00990          MOVE.L     A3,A0          ;Point to string
01000          ADDQ       #1,A0          ;Bump past length byte
01010 .1      ADD.B      #1,(A0)+      ;Increment each byte
01020          DBRA      D1,.1          ;Loop until done

01030          ADD        #1,D5          ;Bump current line no.
01040          ADD        D4,D3          ;Bump line position
01050          CMP        D6,D3          ;Past end of page?
01060          BLE        Ploop          ;No, loop until done.
01070          UNLK      A6              ;Reclaim stack space
01080          UNLK      A6              ;(for two LINKs)
01090          MOVEM.L    (SP)+,D3-D6/A3-A4 ;Restore registers
01100          RTS
01110 *-----
01120 EventLoop NOP                    ;MAIN PROGRAM
01130 Init      MOVE.L     #iPrintSize,D0 ;Allocate print record
01140          OST        NewHandle
01150          MOVE.L     A0,hPrintRec(A5) ;Save handle in hPrintRec
01160          MOVE.L     A0,-(SP)       ;Push it
01170          BSR        PrintDefault  ;Call PrintDefault
01180 *-----

```

```

01190 Style                SUBQ    #2,SP                ;Space for function result
01200                      MOVE.L  hPrintRec(A5),-(SP)    ;Push hPrintRec
01210                      BSR     PrStlDialog            ;Call PrStlDialog
01220                      MOVE.B  (SP)+,D0              ;Pop result
01230 *-----
01240 Job                  SUBQ    #2,SP                ;Space for function result
01250                      MOVE.L  hPrintRec(A5),-(SP)    ;Push hPrintRec
01260                      BSR     PrJobDialog            ;Call PrJobDialog
01270                      MOVE.B  (SP)+,D0              ;Pop result
01280                      BEQ     PrintDone              ;Exit to Finder if cancel
01290 *-----
01300 Spool                SUBQ    #4,SP                ;Space for result
01310                      MOVE.L  hPrintRec(A5),-(SP)    ;Push hPrintRec
01320                      CLR.L   -(SP)                  ;NIL pPrPort
01330                      CLR.L   -(SP)                  ;NIL pIOBuf
01340                      BSR     PrOpenDoc              ;Call PrOpenDoc
01350                      MOVE.L  (SP)+,A4              ;Get pPrPort in A4
01360 *-----
01370 ;Start of page
01380                      MOVE.L  A4,-(SP)                ;Push pPrPort
01390                      CLR.L   -(SP)                  ;NIL pPageFrame
01400                      BSR     PrOpenPage            ;Call PrOpenPage
01410 *-----
01420 ;Draw page here
01430                      BSR     MyDrawPage
01440 *-----
01450 ;End of page
01460                      MOVE.L  A4,-(SP)                ;Push pPrPort
01470                      BSR     PrClosePage           ;Call PrClosePage
01480 *-----
01490 ;End of document
01500                      MOVE.L  A4,-(SP)                ;Push pPrPort
01510                      BSR     PrCloseDoc            ;Call PrCloseDoc
01520 *-----
01530 Print                MOVE.L  hPrintRec(A5),-(SP)    ;hPrintRec
01540                      CLR.L   -(SP)                  ;pPrPort
01550                      CLR.L   -(SP)                  ;pIOBuf
01560                      CLR.L   -(SP)                  ;pDevBuf
01570                      PEA     prStatus(A5)            ;prStatus
01580                      BSR     PrPicFile              ;Call PrPicFile
01590                      MOVE.L  hPrintRec(A5),A0        ;Get printRec handle
01600                      OST     DisposHandle           ;Dispose it
01610                      BRA     EventLoop              ;Go start over
01620 PrintDone          BSR     PrClose                ;Done. Close print manager
01630                      RTS
01640 *-----
01650 ;Data constants
01660 tRect                DATA    /0                ;top
01670                      DATA    /0                ;left
01680                      DATA    /5*72              ;bottom
01690                      DATA    /5*72              ;right
01700 *-----
01710      INCLUDE "Print.Asm"
01720 *-----
01730                      ENDR
01750                      SEG      0,32,VAR.LEN,$20
01770 SEG0
01780 SEG_1                JP      Start,1
01790 END_1
01800 END0
01810                      ENDR
01830                      END

```



Ask Prof. Mac

TextEdit



Steve Brecher
MacTutor Contributing Editor
MacTutor Vol. 1 No. 9

This month, we inaugurate a question-and-answer format for this column. You supply the questions, and I'll do my best to supply the answers. As a print publication, our lead time is too long to help you with those burning 4-in-the-morning, do-or-die questions. But if you've come across something in *Inside Macintosh* that's been puzzling you, or you've been scratching your head about how make the Mac do something you're going to need it to do a few months hence, or you'd like an explanation of some aspect of an example you've seen in MacTutor (or elsewhere) -- write to Prof. Mac, c/o MacTutor.

We start off with a couple of genuine questions from MacTutor readers, followed by a few ringers.

TextEdit

David R. Hunt of Ypsilanti, MI had some questions about TextEdit after reading the Forth Forum in the June, 1985 issue. I'll address these questions with respect to TextEdit in general, rather than specifically with respect to MacFORTH. MacFORTH's interface to TextEdit is described in Section 3 of the Level Two MacFORTH manual.

Q. How big can a TE Record be? How much text can be in a TE Record?

A. First, note that a TERC (TextEdit record) contains information about the text to be edited, but it does not contain the text itself. The text is located elsewhere in memory; the TERC contains a handle to the text. The TextEdit routines maintain both the TERC and the text; you don't have to be concerned with the location of the text in memory.

Background on TextEdit

To preserve the Mac way of doing things, Apple defined a consistent approach to dealing with the entry, editing and display of text information that would operate cleanly with the graphics-oriented hardware design built around quickdraw. That approach is a collection of ROM toolbox routines that handle simple text editing, and is known collectively as "TextEdit". Think of it as a mini-editor built into the ROM. The mystery of the Mac approach to text editing lies in the fact that the Mac does not have a traditional character generator. Everything is displayed on the screen and within the system in bit-mapped graphics via quickdraw. Thus each character of text is a graphics symbol made up of screen bits. This both provides the flexibility of the Mac in generating different fonts, sizes and orientations while at the same time making the control of

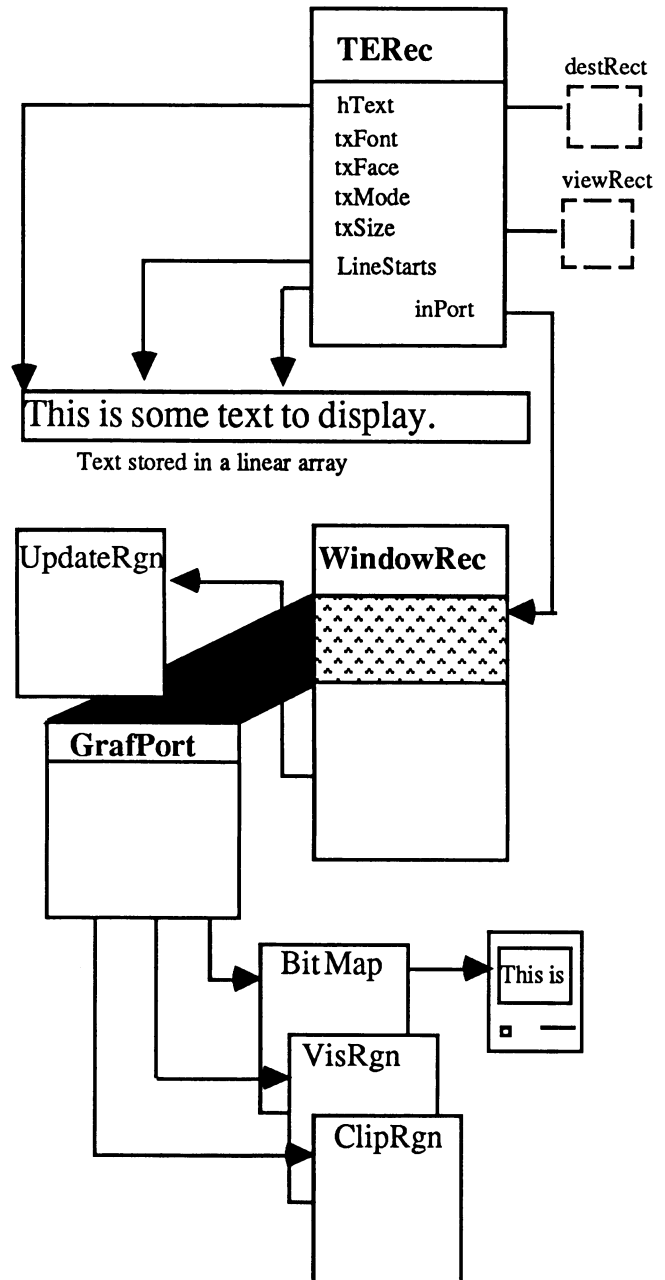


Fig. 1 TextEdit Data Structures

text information more complex. The text edit record structure is a method of isolating the text itself and the information about how the text is to be displayed in two different data structures. The text is stored in a coded linear array, while all the information about how that array of text should look is stored in the text edit record. If any of the text changes style or font, that text must be associated with another text edit record to describe how it is to be displayed. Thus if a document has many font changes, more and more text edit records must be generated for each section of text. (See Fig. 1)

TextEdit Capabilities

- (1) Insert new text
- (2) Delete text via the backspace key
- (3) Select text with the mouse
- (4) Scroll text within a window
- (5) Cut and Paste of text between the clipboard
- (6) Display of selection bar and text highlighting
- (7) Word wrapping of text
- (8) Left, centered or right justified text

TextEdit Limitations

- (1) Only one font per text edit record
- (2) No fully justified text
- (3) No tab support
- (4) Only simple cut and paste

Clipboard Support

The new phone book version of *Inside Macintosh* points out that cut or copying text with TextEdit places that text in a special TextEdit scrap that is only used within TextEdit. A different scrap is used to support the clipboard for moving text between applications. Documentation for the routines that move text into this second scrap has now been added to *Inside Macintosh*.

The important components of a TERec are shown on page 48 of the June issue; the complete TERec structure is detailed in *Inside Macintosh*, /TEXTEDIT/EDIT.2, p. 9. For a complete discussion of how text editing interfaces with windows, see Bob Denny's C column in the May 1985 issue of MacTutor.

And now Back to the Question...

At the end of a TERec is a variable-length array called `lineStarts`; there is one 16-bit unsigned element in the array for each line of text. Each element contains the position (offset) within the text of the start of a line. The number of lines (and hence the number of elements in the `lineStarts` array) is contained in a 16-bit signed field of the TERec called `nLines`. The maximum value of `nLines` 32,767. Thus, the maximum number of lines in the text, and of elements in the `lineStarts` array, is 32,767.

Since each element of the `lineStarts` array is an unsigned

16-bit quantity, each element has a maximum value of 65,535. So the position of the text's last line can't be more than 65,535 characters away from the beginning of the text.

Therefore, the text can have at most 32,767 lines, and the number of characters preceding the last line can be at most 65,535. The size of the TERec itself is the total of the sizes of its fixed-length fields, 96 bytes, plus the size of the `lineStarts` array (2 bytes for each line).

Q. Can I have more than 1 TE record in a window?

A. Strictly speaking, the TERec is not "in" a window; it is merely a data structure describing some text (located elsewhere in memory) and describing where and how the text is to be displayed. In fact, TextEdit does not "know" about windows at all. It displays the text within a `grafPort`; the `grafPort` is not necessarily one with which a window is associated (although in practice it almost always is a window's `grafPort`).

The `grafPort` in which the text associated with a TERec will be drawn is the `grafPort` that was current at the time the TERec was created. Its address is stored in the `inPort` field of the TERec. TextEdit locates the display area for the text within the `grafPort` as specified by a destination rectangle whose coordinates (local to the `grafPort`) are contained in the `destRect` field of the TERec. The `destRect` is specified by the programmer as a parameter to the `TENew` routine (which creates a TERec).

There can be more than one TERec associated with the same `grafPort`, and hence with the same window. In this case, each TERec would have a different, non-overlapping `destRect` within the `grafPort` (or window).

Q. What happens if I update a window with multiple TE records?

A. Remember that updating is not an automatic process. Your application is notified that an update event has occurred, which means that a window needs to be updated. Generally speaking, you must do the actual updating, or redrawing of the contents of the window, yourself.

However, TextEdit makes it easy to update the text portion(s) of a window. You call `TEUpdate`, giving it a rectangle in which to (re)draw the text, and a handle to the TERec associated with the text. Usually, the rectangle you'd pass to TERec would be simply the `portRect` of the window. If there are multiple TERecs associated with a window, you'd call `TEUpdate` once for each TERec. All of the calls would be bracketed between calls to `BeginUpdate` and `EndUpdate`, which is the way window updating is done generally (regardless of whether text is involved).

Q. In my application, I want to issue text to a TE Record. How do I do it?

A. As noted above, the text itself is not contained in the TERec; rather, a handle to the text is contained in the TERec.

If you have some pre-existing text which you want to associate with a Terec (replacing any text already associated with it, if any), you call TEsSetText. The parameters to TEsSetText are the address of the text, its length, and a handle to the Terec. This establishes the connection between a linear array of text, the text edit record that describes how and where the text is to be displayed, and the screen area where a portion of that text may or may not be visible to the user.

To insert some new text into text already associated with a Terec, you call TEInsert, passing to it the address of the text to be inserted, its length, and a handle to the Terec. The insertion will occur just before the current selection range.

If the current selection range is an insertion point (indicated on the screen by a blinking vertical bar (or caret), you can insert a single character there by calling TEKey. The parameters to TEKey are the character, and a handle to the Terec. TEKey is so named because usually the character is one that was typed by the user and obtained from the event record associated with a keyDown event, but this is not necessarily the case: you can specify any character. If, when TEKey is called, the selection range is indeed a range (indicated on the screen by white-on-black text) instead of an insertion point, the range will be *replaced* by the character.

Q. Do I append the TE records together, or leave them separate?

A. I hope it's clear from the above that Terecs cannot be concatenated. Each is a separate structure, describing how and where to display text associated with it.

Q. My worst problem: TE records are utterly unlike text processing as I have previously known it on mainframes. Could you show me how to make this transition?

A. I hope the preceding answers have helped some. You may be suffering from an embarrassment of ROM riches: in most systems, a programmer must take care of all the details of text editing himself. But on the Mac, the TextEdit facility does almost all of the work for you. Figure 1 at the beginning of this article may help to present the big picture.

Mac Pascal Bug

In last month's Letters section, Jack Cassidy of San Diego provided a simple Macintosh Pascal program that bombs for no apparent reason. Our editor promised that I'd look into the problem.

Q. Why does the following program cause a system error (ID=02)?

```
program SystemBomb;
type
  count=(one,two,three,four,five);
var
  simple: array[count] of integer;
begin
```

```
  simple[five]:=31;
end.
```

A. Unfortunately, Jack has run into a bug in Mac Pascal: it does not handle array references correctly where the subscript is an enumerated type. A workaround is to do away with the enumerated type and replace it with a collection of constants. This retains some of the readability provided by enumeration, at the expense of some coding tedium. For example:

```
program ShouldWork;
const
  one = 1;
  two = 2;
  three = 3;
  four = 4;
  five = 5;
var
  simple: array[one..five] of integer;
begin
  simple[five]:=31;
end.
```

Note that in the program that bombs, ord(one)=0, but in the workaround version ord(one)=1. This makes no significant difference; I just wanted to avoid writing the apparently illogical "one = 0; two = 1;" etc. If the names of the constants were not those of numbers, then the first one could be given the value 0 without risk of confusion.

MaxApplZone

Now here's one of my own questions. The MaxApplZone procedure is marked "No trap macro" in Inside Macintosh, implying you can't use it from Assembly.

Q. How would I write a MaxApplZone in MDS assembler?

A. See the code in Figure 2 on the next page.

Long Division

The fact that the MC68000 hardware divide instructions handle only 16-bit divisors and return 16-bit results presents another question:

Q. How would I do 32-bit unsigned division in MDS assembler?

A. See the code in Figure 3 on the next page.

Include SysEqu.D

Figure 2

```
;
; "MaxApplZone expands the application heap zone to the application
; heap limit without purging any blocks currently in the zone. If
; the zone already extends to the limit, it won't be changed."
; --Inside Macintosh, specification of MaxApplZone
;
MinBlkSize Equ 12           ;minimum physical size of a block
;
MaxApplZone:
    Move.L    ApplLimit,A0    ;A0 = application heap limit (pointer)
    Move.L    HeapEnd,A1      ;A1 = addr of current zone trailer
    Move.L    A0,D0
    Sub.L     A1,D0           ;D0 = size of unused space (incl. trailer)
    MoveQ     #MinBlkSize,D1  ;D1 = minimum physical size of a block
    Cmp.L     D1,D0           ;unused space < minimum size of a block?
    Blo.S     @0              ;if so, forget it, can't expand zone
    Move.L    A0,HeapEnd      ;update HeapEnd to heap limit
    Move.L    D0,(A1)         ;unused space := free block (header:=size)
    Move.L    ApplZone,A1     ;A1 --> zone header
    Move.L    A0,(A1)         ;bkLim field of header := heap limit
    Move.L    D1,(A0)         ;make new zone trailer (free, size=min)
    Add.L     D0,zcbFree(A1)   ;update count of free bytes in zone
    @0 Rts
```

Figure 3

```
; On entry:
;   D0 = 32-bit unsigned divisor
;   D1 = 32-bit unsigned dividend
; On exit:
;   D0 = 32-bit unsigned quotient ($FFFFFFF if divisor = 0)
;   D1 = 32-bit unsigned remainder (dividend if divisor = 0)
;   D2 = $FFFFFFF
;   D3 = D0
;
LongDiv:
    MoveQ     #-1,D3          ;init complemented quotient
    MoveQ     #-1,D2          ;init complemented shift counter
    Tst.L     D0              ;divide by 0?
    Beq.S     @6              ;yes, quotient=max magnitude,
                                ;remainder=dividend
    @0 Cmp.L   D0,D1           ;dividend < (shifted) divisor ?
    Blo.S     @1              ;br if so
    Add.L     D0,D0           ;shift divisor left
    DBcs     D2,@0            ;if didn't shift out, update counter
                                ;and loop
    Rorx.L    #1,D0           ;shifted out: undo shift...
    Not       D2              ;...get true shift count
    Bra.S     @3              ;...into loop (bypass first undo of
                                ;shift)
    @1 Not     D2              ;get true shift count
    Bra.S     @5              ;into loop
    @2 Lsr.L   #1,D0           ;undo a divisor shift
    @3 Sub.L   D0,D1           ;subtract (shifted) divisor from
                                ;dividend
    Bcc.S     @4              ;skip if no borrow (X=0)
    Add.L     D0,D1           ;restore (and set X)
    @4 Addx.L  D3,D3           ;shift old quotient, add
                                ;complement of new bit
```

```
@5 Dbra     D2,@2            ;loop until divisor shifts all undone
    Not.L    D3              ;uncomplement quotient
    @6 Move.L  D3,D0          ;return quotient in D0
    Rts
```

MDS Asm Bug

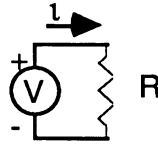
Q. Asm gave me an error message for CMP (A0)+, (A1)+. After I changed the opcode to CMPM, my program bombed later with an address error. Why?

A. The release version 1.0 of the MDS assembler does not automatically generate CMPM as it should when CMP is coded with two auto-increment operands; you must explicitly tack on the "M". Further, if you code CMPM, it generates a CMPM.B; to compare words, you must code CMPM.W.



The Electrical Mac

Mac Memory Explained



Jeff Mitchell
MacTutor Contributing Editor
MacTutor Vol. 1 No. 9

The Macintosh has been unique among personal computers in that it was the first computer introduced with a minimum of 128K Bytes of RAM that was immediately criticized for not having enough. The Fat Mac has quieted that criticism somewhat, and there are rumors of even more memory upgrades coming. [Experience has now shown that the Mac design required a minimum of 512K RAM and 800K dual floppy drives to be really practical and friendly to the user. We have the memory now but the disk bound nature of the new application packages is making disk juggling a full time Mac occupation! Say, didn't the old "Flippy" Lisa drives support 800K? That early Mac design was right-on capacity wise. -Ed.]

In this article I will describe the basic internal architecture of dynamic RAM, explore the design tradeoffs that have led to the memory configuration and the data access techniques employed in the Macintosh, and describe the memory map with its various screen, sound, and disk speed buffers.

More dynamic RAMs are sold than any other type of semiconductor device in the world. This is due to two major factors. First, the demand by computer manufacturers for inexpensive, high density memory. Secondly, semiconductor manufacturers use dynamic RAMs as a testbed for their IC process technology. The very regular structure of one memory cell repeated thousands of times, plus the very high demand for these devices allow the manufacturers to develop new, smaller device geometries with reasonably low risk and then use the large demand to speed the improvement of production yields. Once the technology has been refined, it is used to develop other products. Semiconductor manufacturers have admitted to

being in the dynamic RAM market not necessarily to make money, but only to break even, in order to have this development vehicle available to them.

Each bit in a dynamic RAM consists of a transistor and a capacitor. This is why such very high densities can be attained, since each individual cell is so small. In contrast, each bit in a static RAM consists of four or six transistors, arranged to form a latch. Since each cell in a static RAM is about 4 times as large as in a dynamic RAM, static RAMs lag about one generation behind dynamic RAMs in density (e.g. 64K static RAMs became available about the time 256K dynamic RAMs were being announced).

With dynamic RAMs, the capacitor used as the memory element is not perfectly insulated, allowing the charge to leak off over time. Thus it may at some point actually change state unless the charge is restored. The charge can be restored by performing either a read or write operation on the cell. This process is called refreshing and must be performed on all cells every 2 milliseconds with most 64K RAMs, or every 4 milliseconds with 256K RAMs.

Internally, the memory cells are arranged in a matrix of rows and columns. The intersection of a row address and a column address selects a single cell. Dynamic RAMs are usually packaged in a 16 pin package, which means the addresses must be multiplexed due to pin limitations. First the row addresses are presented along with the row address strobe (RAS), then the column addresses and the column address strobe (CAS) some time later. A very important advantage to the matrix configuration is that when a particular row is accessed, all the cells on that row are refreshed; each cell does not have to be refreshed individually.

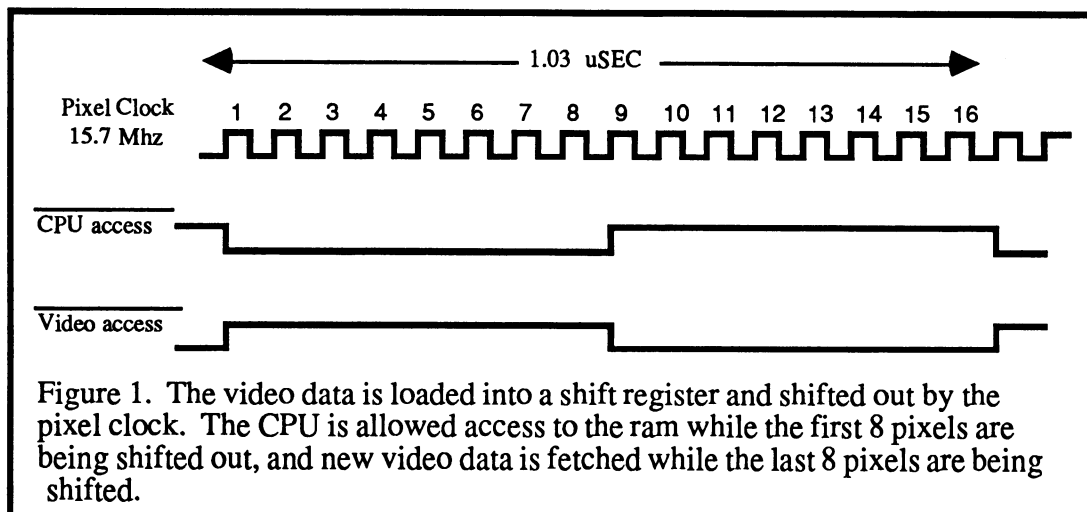


Figure 1. The video data is loaded into a shift register and shifted out by the pixel clock. The CPU is allowed access to the ram while the first 8 pixels are being shifted out, and new video data is fetched while the last 8 pixels are being shifted.

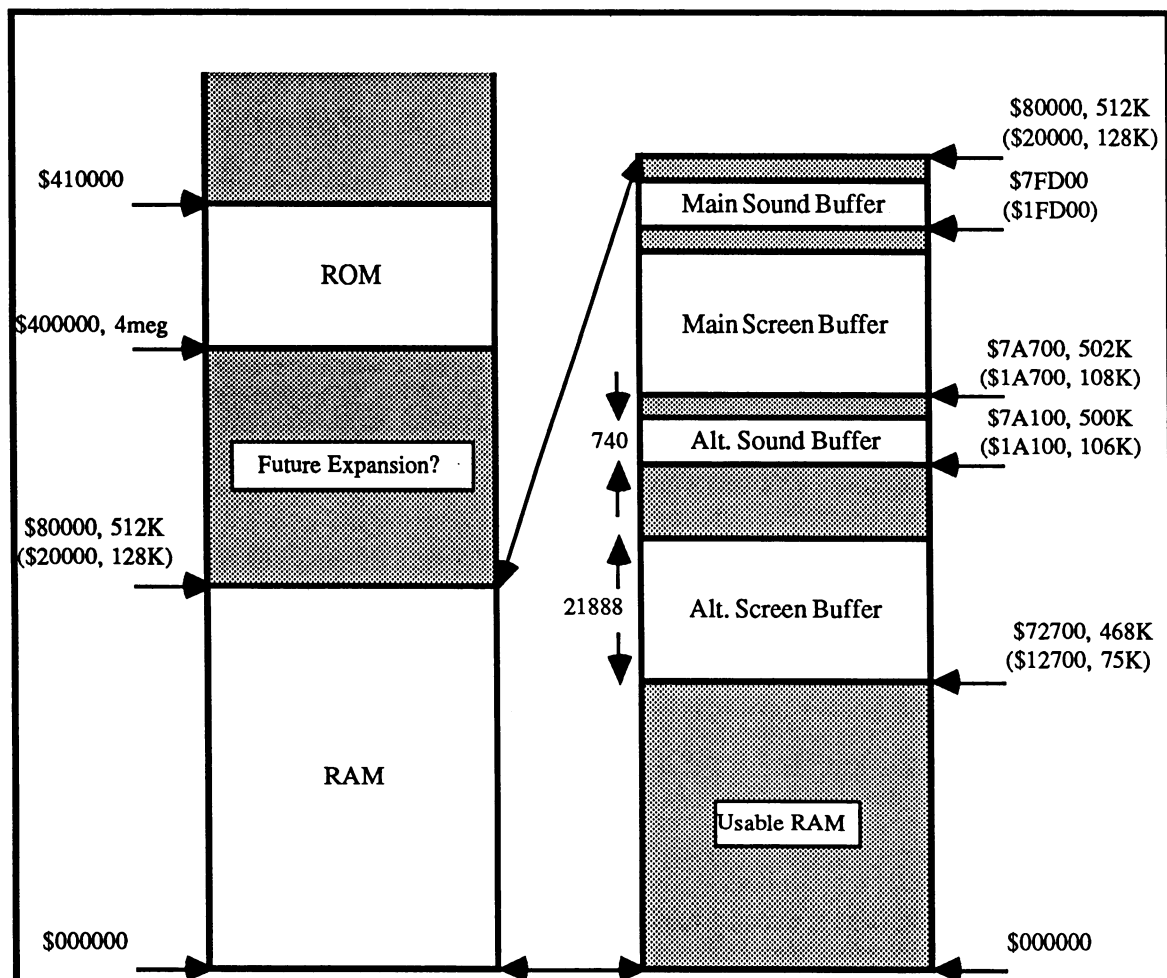


Figure 2. The Macintosh Memory Map

Notes: The 128K Mac has only 75K of continuous RAM for application programs (heap and stack use rest).

The design of the 512K Mac shows it can continuously address up to 4 Meg before hitting ROM, but the video and sound buffers at the top of the 512K space would have to relocate upwards.

The RAM in the Macintosh is shared between the processor and the screen, rather than the screen having a separate RAM area of its own. This is both a blessing and a curse. The blessing is that the screen accesses are frequent enough that there is no need for special circuitry to keep the RAM refreshed. The curse is that the processor is denied access to the RAM about 34% of the time while the screen is being refreshed.

The display RAM is bit-mapped, meaning that for each pixel on the screen there is a corresponding bit in memory. As the electron beam sweeps across the CRT, it is turned on or off depending upon the state of the bit assigned to that pixel (1 = off, or black; 0 = on, or white).

How can both the screen and the processor use the same RAM? The Macintosh memory design uses a standard technique called interleaving, where the screen and the processor alternate having access to the memory. Since the

screen cannot be denied access, the processor must wait if there is a potential memory conflict (Figure 1). This type of RAM configuration is known as dual-port, where there are two independent and (seemingly) non-conflicting paths into the same memory area.

There are actually two screen buffers in RAM. The main buffer located at \$7A700 (\$1A700 in a 128K Mac), and an alternate buffer located at \$72700 (\$12700). Port A bit 6 of the 6522 VIA (Versatile Interface Adapter) is factored into the address decoding logic for the screen RAM and determines which buffer will be displayed. Figure 2 graphically describes the memory map and Figure 3 is a table showing the address location and effect on the memory map of VIA Port A.

In addition to the screen memory, the RAM also has a dedicated area set aside for the sound/disk speed buffer. This is accessed at the end of every horizontal retrace period and supplies an 8 bit value to the sound generator PWM (pulse

width modulator) and a 6 bit value which is used to control disk motor speed. Again, as with the screen RAM, there is a main and alternate buffer at \$7FD00 (\$1FD00) and \$7A100 (\$1A100) respectively.

The memory access scheme for the ROM is much simpler than that of the RAM, since only the processor needs access to it. It is located at \$400000 in both 512K and 128K Macs, and extends to \$40FFFF (64K Bytes). ROM accesses always occur at full speed, with no wait states.

The ROM upgrade, rumored to be out this fall, expands the size of the ROM to 128K. It is interesting to note that the two ROMs are the only chips that are in sockets, suggesting that Apple foresaw this eventual need to revise and expand the ROMs. Unfortunately, any additional upgrades beyond 128K can only enhance and/or debug the ROM code unless there is (once again, as with the 256K RAMs) a new circuit board, since the ROM sockets cannot accommodate devices larger than 64K x 8 (there are two ROMs, making 128K total).

As with the screen and sound/disk speed buffers, there is a bit which affects the mapping of the entire RAM and ROM. The overlay bit (Figure 3) remaps the ROM to \$000000 and the RAM to \$600000 on reset. This is because the first four bus cycles after a reset, the 68000 fetches the supervisor stack pointer and the program counter from memory location \$000000. This is where the RAM resides, but it contains invalid data at reset, so the ROM is placed there temporarily. After the exception tables are initialized and everything is set up, the overlay bit is cleared and the memory map returns to its normal configuration.

The electrical design of the Macintosh is elegant, although following in Apple tradition, is somewhat simplistic. For example, the address decoding for write operations to the 8530 SCC (Serial Communications Controller) relies on the 68000 to place the 8 bits of data on the high byte of the data bus when it is actually doing a write to the low byte. This it does, but Motorola warns that this is "a result of current implementation and may not appear on future devices." Apple is obviously gambling that Motorola

ADDR/BIT #	DESCRIPTION
\$EFFFFE	VIA Port A
0,1,2	Sound Volume (0=off, 7=loud)
3	Sound pg. 2 (0=pg. 2)
4	Overlay bit (1=overlay)
5	Head Select
6	Video pg. 2 (0=pg. 2)
7	SCC wr/req

Figure 3. VIA port A bit map - bit 7 is an input, all others are outputs.

won't change the implementation and that any parts they may buy from another source will perform identically. The benefit here is that the fewer the number of parts, the less there is to break and the cheaper the product is to manufacture.

Stay tuned for more Mac hardware, including future Apple Bus construction projects! If you have questions or ideas for future hardware topics, please share them with us. Write "The Electrical Mac" care of MacTutor.



Debugging Techniques and TMON Review

I was just reading my Volume 1, No. 4 MacTutor and came across William Gladnick's letter asking about tips, utilities, and debugging techniques. OK, I'll bite!

MacsBug Snoops the ROMS

As far as useful utilities go, first, get a 512K system. Then get MacNOSY and TMON (the latter is from TMQ software). These, plus a good assembler (MacASM, in other words) are all that you really need to do serious Mac hacking.

As for tips, here's one: the Mac ROMs are buggy, and nearly everyone knows it. What's not so obvious is how to use MacsBug to trace through the ROMs step by step (rather than treating the A-traps as a single instruction). Wouldn't it be nice to be able to trace one instruction at a time while in a ROM routine? OK, here's how: write your application (I assume that you are using one assembler or another). The first instruction should be:

```
LABEL BRA LABEL
```

followed by your program. Assemble the program in the usual way. Install MacsBug in your system. Double-click your program's icon. The system will seem to hang, since you told it to BRanch Always to LABEL, which is where the branch is. In other words, your program is in an infinite loop. Press the interrupt button on your Mac. MacsBug will be entered with the PC pointing to your BRA instruction. Bump the PC by two to point past the BRA. Now start tracing your program.

How to Trace the ROM Routines

As you are tracing or stepping through your code with MacsBug, the disassembly will eventually say something along the lines of "OSTRAP A219" or "TOOLBOX A912" or what have you. STOP! DO NOT TRACE OR STEP! Now, change the value in A7 downward by six bytes. In other words, if A7 is now 00123AC7, change it to 00123AC1. This is to simulate the stack setup caused by the 68000 during exception processing.

Now, look at the current PC value. Let's say it's 00004E56. Store this address at A7+2 using the SM command, like this:

```
SM 00123AC3 00 00 4E 56
```

Now type PC 401018. You may now continue tracing or what have you. *Essentially, you have done manually what*

the 68000 would have done automatically upon encountering the A-trap. 401018 is the address of the Macintosh A-trap dispatch routine. Observe the code as you trace through it; it's quite interesting. For example, the main difference between OSTRAPs and TOOLBOX traps is that for OSTRAPs, which require an I/O buffer address in A0 and return a status code in D0, A0 and D0 must not be saved on the stack across the actual code. Apple accomplished this by setting bit 11 in all of the TOOLBOX instructions (in other words, the second hexadecimal digit of a TOOLBOX trap will always be at least 8). The dispatch code at 401018 examines that bit to determine whether or not to save A0 and D0 (and possibly some other registers) on the stack.

More importantly, you can watch as the dispatch code determines via a lookup table where to go in the ROMs, and you can trace through those high-power ROM routines that we've all come to know and love. Now you can see for yourself exactly how a window is opened, or what have you.

The ability to trace through the ROMs, coupled with MacNOSY's capability of outputting labels, as opposed to addresses, makes for a relatively simple way of learning how the Mac works right down at the nuts and bolts level - which is right up MacTutor's alley, right, guys?

May all of your hacking - er, excuse me, I mean programming - be bug free!

TMON Thoughts

Some of you may have seen TMQ Software's ad in MacTutor lately. They're pushing a product called TMON, which at first made me laugh - TMON was the name of the first machine language monitor I ever actually laid my mitts upon; it was for my (at the time) tape-based Model I TRS-80. You can see why I laughed.

Nevertheless, TMQ promised some interesting things, like tracing ROM and setting up to seven breakpoints. Also, they claimed that TMON was a multiple-window monitor/debugger. Just the idea that a debugger could be multiple window boggled me. The thing must be *huge*. Especially if it can disassemble, do hex dumps, heap displays, register displays, and so on simultaneously.

The price looked pretty reasonable for all of this: the ad asked for \$100.00, which is less than I paid for MacASM, my assembler of choice. I finally broke down and ordered TMON from TMQ. It arrived a few days after I ordered it; if you're willing to pay for it, TMQ will ship UPS blue label, which is how I prefer to work.

After using TMON and configuring it and reconfiguring it and debugging some of my own code and tinkering around

with some code that I didn't write, I can make the following observations.

Love at First Byte

First, if you don't already own a copy of TMON, run, don't walk, to get your checkbook and order it *now*. If you are a serious developer (and if you are reading this, chances are pretty good that you are) you should be using TMON. Use TMON once and you'll never boot MacsBug again. I promise.

The ads really don't do it justice. They don't tell you about the interactive assembler/disassembler; they don't tell you about the fact that dump and disassembly windows can be "anchored" to registers and are continually updated; they don't tell you that TMON allows access to any online resource file; they don't tell you about the customizable user area; they don't tell you about the sophisticated label support, or the screen buffering, or the rich implementation of interrupt button functions... The list goes on.

Think of TMON as MacsBug and MacNosy combined, with some things thrown in that neither MacsBug nor MacNosy was intended to do. To be fair about it, since MacNosy is a disassembler and not a debugger, there are some features that MacNosy has that TMON doesn't. I recommend both of them if you are serious about Mac programming.

TMON's fast window system will impress you, as will its implementation of menus. It takes a little getting used to; both windows and the menubar are handled a little differently from "normal" Mac windows and menus.

Mouse Unfreeze

The menus are Dump, Asmbly, Brkpnts, Regs, Heap, File, Exit, GoSub, Step, Trace, Options, Num, User, and Print. Another feature that TMON implements via a command keystroke is called Mouse Unfreeze. Yes, you read right - if your program crashes and your mouse freezes up but the keyboard is still listening, you can unfreeze the mouse. Wild stuff.

Dump opens a window which is

initially a dump starting at address 000000. There is a blinking cursor on the first line. As long as the cursor is on the first line, the address can be changed. You can type in any expression, label, or register to anchor to (as well as a special virtual register called V). The window will be instantaneously updated to show the new dump. More importantly, if you anchor to a register (say A0), whenever the contents of A0 changes the window will be updated to reflect the change. You won't believe how useful this is until you see it.

Disassembler Anchored to the PC Counter

'Asmbly' is the interactive assembler / disassembler. When you open this window, you get a disassembly starting at 000000, which doesn't make much sense, since there is no valid code there. You'll probably want to anchor the disassembly window to the PC. Like the dump window, when you anchor the disassembly window to a register it gets updated every time the register changes. Anchoring the disassembly window to the PC causes the window to be updated every time you step, trace, gosub, or exit to the program (actually, since exit actually transfers control to the program, the window will not be updated unless control returns to the monitor).

TMON normally displays A-traps as labels, not numbers, although this can be changed in order to conserve space. TMON also supports user-defined label tables. The TMON master disk includes a file called System.MAP which is a label file of all of the low memory equates on the Macintosh. This MAP file can be loaded into TMON so that features of TMON that support labels (nearly all of them, in other words) can now refer to low RAM areas (with labels like CurApName and so on) by name instead of address. This aspect of TMON, unfortunately, requires 512K.

TMON includes sophisticated customization support that allows you to fine tune TMON to your needs.

TMON is quite capable of running on a 128K machine, and contrary to what you might think, it does not have to be hopelessly crippled in order to do so. I can use A-trap names and, with a compressed screen buffer of 4K, do some meaningful debugging. A 512K Mac sure would help, though.

TMON has an alternate register save area. This is interesting. You can come close to a routine that you think will crash, save the registers, execute the routine, and if it crashes (an exception lands you back in TMON), you can reload the registers and try again, and so forth. Kind of like being in suspended animation.

We all know that the interrupt button on the Mac is good for something. TMON knows it, too. Pressing the button from an application will get you into TMON, just like it does with MacsBug. However, there are several special-purpose keystroke-interrupt combinations that TMON has over MacsBug.

Pressing interrupt while holding down the command key restarts the monitor. This is considered a drastic step to be taken only when all else fails. This often has the effect of mulching part of TMON. The nice thing is that TMON is constantly checking itself, and if it thinks it's been damaged, it will say so.

Cmd-Option-Interrupt!

If that doesn't work, try command - option - interrupt. This does a really total monitor reset, which makes it definitely last-resort material.

Now for the interesting one. The user area supplied with TMON contains a function called Trap Signal. The manual calls the Trap Signal function a "smart interrupt," which is a pretty apt description. Trap Signal allows you to specify a trap or range of traps to apply this function to, as well as an optional address or address range. Once you've done that, go ahead and execute your program.

Nothing happened, right? Your program should be whizzing merrily along. Now press interrupt while holding down the option key. This

activates the trap signal, which won't go into effect until your trap and address range criteria are met. That's why it's called a smart interrupt: it doesn't stop until it hits the right trap/PC location. This allows you to interrupt your program at a specific place rather than going into it blind. A good trap to put a smart interrupt on would be `_GetNextEvent`, since it tends to be at the main loop of any Mac application.

Trace uses the Trace bit

TMON's trace, step, and GoSub features are *very* nice and *very* powerful. TMQ Software saw fit to do what no one else did: they use the trace bit of the status register to force a trace interrupt, which the monitor traps. This is the essence of tracing under TMON. The trace function even allows you to watch as the A-trap dispatcher finds the address of the A-trap's code and executes it. This means that my little note concerning tracing the ROMs under MacsBug doesn't apply here; TMON's trace does it automatically. Step in TMON is exactly like trace except that A-traps are treated as a single instruction. GoSub is like step except that subroutines (executed by JSR or BSR) are executed in full without tracing.

The heap windows are something else. Not only do they show what blocks exist and whether the block is locked, purgable, relocatable, and what have you; they also show what is in the block (to the best of TMON's ability). You can see at a glance which block contains CODE segment number one of your program, which block contains the font that is currently in use, which blocks contain resources pertinent to the application, and so on. You can also examine the system heap in the same manner. A heap window can be made to switch from application heap to system or vice-versa by hitting the TAB key (which puts the cursor at the top of the window) and hitting the RETURN or ENTER key.

The file window is extremely useful. It shows the contents of all currently open resource files. For those resources that are in memory, their

handle is given. For those that haven't been loaded from disk yet, "Nowhere" is displayed. Resources of any type and ID can be loaded using the standard user area LoadRes utility.

If a window is too limited to show you as much as you would like of what it is you're looking at, you can probably print it with the user area print utility. You can specify dump printing, disassembly printing, file printing, or heap printing. For dumps and disassemblies you must specify an address range. For files TMON needs the resource file number, and for heap prints it needs 0 for the system heap and a non-zero value for the application heap.

There are lots of other handy features, too, such as block moves, block compares, block fills, check-summing of memory, label table support, a very efficient heap scramble, a find utility, the ability to show the application's screen, and the ability to completely reset the system.

User Extensible via MDS

To make matters even better, the author includes the source code to the entire default user area on the TMON disk. The point is that sophisticated Mac programmers can write their own user area utilities that can either replace or augment the ones included with TMON. The source is in MDS format, and a link file is included to link the resulting object code into TMON. Obviously, you must have MDS for this to be of any use. Frankly, I think it'll be a long time before anyone comes up with a way to improve upon TMON's already fantastic user area routines.

That about wraps it up. My idea of an ideal Mac debugging setup is a 512K Mac running TMON. Use the normal default user area, then add in the System.MAP file. This will allow TMON to refer to low memory areas by name (which is handy if you're doing ROM disassemblies in particular). Now you can *really* debug! Goodbye, MacsBug; hello, TMON!



Ask Prof. Mac

Screen Sleep in Assembly



Steve Brecher
MacTutor Contributing Editor
MacTutor Vol. 1 No. 10

BASIC Sort

David Smith, Editor of MacTutor, laments, "I wrote an MS BASIC program that sorts an array of some 600 items (names). *It takes over an hour!* Absurd. It's a simple bubble sort."

Q. How much time can be saved by an alternate sort algorithm in BASIC?

A. The bubble sort is a notoriously poor performer except on very small arrays. Sorting algorithms and their performance are a well-studied subfield of computer science. Two clever algorithms that have become popular outside of academia are ShellSort and QuickSort; I'll discuss QuickSort because I'm more familiar with it.

In its pure form, QuickSort is a simple recursive algorithm that uses a "divide and conquer" technique. Consider the elements of the array to be sorted as laid out horizontally, so we can talk of the left and right portions of the array. If it is the case that (1) some element of the array, called the *partitioning* element, is already in its final position; (2) all the elements to the left of the partitioning element are less than or equal to the partitioning element; and (3) all the elements to the right of it are greater than or equal to it; then all we need to do is to sort the subset of elements to the left of the partitioning element and then sort the subset to the right. QuickSort consists of a simple sub-algorithm to set up the partitioning as specified above, followed by two recursive calls to itself to sort the left and right partitions.

While QuickSort is very simple and a good performer in many cases, it has some drawbacks: recursion is often inefficient or (as in the case of MS BASIC 2.0) not available; and in some instances -- e.g., when the data happen to be already in the proper order before sorting -- QuickSort performance

degenerates. (Yes, the original QuickSort routine actually "prefers" randomly-ordered data!) So, improvements have been made -- at the cost of adding some complexity to the algorithm.

The optimized QuickSort algorithm that I have ported to MS BASIC was developed by Robert Sedgewick from C. A. R. Hoare's original algorithm. It makes the following improvements: (1) it uses a more complex partitioning sub-algorithm to enhance performance and avoid worst-case degeneration; (2) it avoids recursion; and (3) it uses a simple algorithm called an *insertion sort* when the sizes of the left and right subarrays become "small"; the insertion sort is more efficient than QuickSort on small arrays.

The MS BASIC routine, including references to a paper and book by Sedgewick, is shown in Listing 1. The routine takes about a minute to sort 600 strings having an average length of 18.2 characters. The Sort subprogram could be easily adapted to sort a numeric array instead of a string array by removing all occurrences of the "\$" character.

TextEdit & Dialogs

Last month, we provided an overview of TextEdit, the Mac's built-in text processing facility. Application programmers are not the only ones to make use of TextEdit facilities; TextEdit is also used by the Dialog Manager to display and handle the editing of text items in dialog boxes. An anonymous caller to MacTutor's offices asked:

Q. How can I change the font size used in dialog editText items? Simply changing the txSize field in the dialog's TEREc doesn't seem to work.

A. Changing the value in the txSize

field *will* work -- but the timing is crucial.

To simplify the discussion, let's assume for the moment that we're talking about modal dialogs. Assume that the dialog contains a variety of items including controls and statText items as well as editText items. We want one or more of the editText items to echo the user's keystrokes in a font size other than the default.

We create the dialog with NewDialog or GetNewDialog. At this point the dialog's window and controls are drawn (although they may not be *visible* if we've made the dialog invisible). None of the other items are drawn at this time; instead an update for the dialog window is generated. Typically, at this point we'd call ModalDialog in a loop, and after each call check the itemHit value to see what we must do to deal with the item. When ModalDialog is first called, it will handle the update event that was generated when the dialog was created, i.e., it will draw the dialog's items.

The Dialog Manager uses TextEdit not only to handle editText items, but also to draw statText items. And it uses only one TEREc for all text items in a given dialog, changing the TEREc's contents as required as the current item changes. Therefore, if before calling ModalDialog we get the address of the dialog's TEREc from the textH handle in the DialogRecord and change the txSize field in the TEREc, then all the statText items will be drawn in that size. We'll have achieved our purpose of changing the size of editText key echos, but with an unpleasant side effect upon the statText items.

What we should do, then, is to write a *filterProc* and pass its address to ModalDialog. Our filterProc will be able to inspect each event before ModalDialog acts upon it. If the event is a keyDown (key press), then we can

go ahead and change the txSize field in the dialog's TEREc. Optionally, before doing so, we can check the editField value in the DialogRecord to see whether we want to change the size in the specific editText item that is current. If we've previously changed the size, but the current editText item is not one for which we want it changed, or the event is not a keyDown, we should change the size back to the default. The default value in the txSize field is 0, so all that's required to undo our change is to clear txSize.

For modeless dialogs, we can change the txSize field just before calling DialogSelect if the event we're passing to DialogSelect is a keyDown. Whatever our filterProc was doing for a modal dialog, for a modeless dialog we can do between calling IsDialogEvent and calling DialogSelect.

Finding a Finder

Another anonymous caller to MacTutor (I'd like to give credit to questioners, but our Editor forgets to take their names) happened to ask questions about Finder and MiniFinder that I'd just been researching on my own behalf. In particular, I was having trouble figuring out how MiniFinder overrides the normal process that occurs when a program quits. Thanks to Bill Steinberg, Sysop of the Mac User's Forum on CompuServe MAUG for pointing out the INIT 4 resource described below (MAUG is a trademark of MCU, Inc.; Bill Steinberg is not a trademark and may be freely abused.)

Q. How does the Mac know to run Finder when an application quits? And if MiniFinder is installed, how does it know to run that instead?

A. There is a system global at address \$2E0 called FinderName. It is 16 bytes long, and its contents are interpreted as a Pascal-format string, i.e., length byte followed by characters. Thus, the name stored there can be at most 15 characters long. At boot time the string "Finder" is stored in FinderName.

An application quits by invoking

the ExitToShell trap (or by executing an RTS instruction, which invokes ExitToShell indirectly). ExitToShell passes the FinderName address to the Launch routine as a pointer to the name of the application to be launched. As noted, normally that name is "Finder," so Finder is launched.

If a program wants to replace Finder as the default "shell," all it need do is copy the filename of the replacement to FinderName (in Pascal string format). Henceforth, when ExitToShell is executed, the replacement program will be launched. Note that the user will never be able to get back to Finder unless the replacement program gives him a means to do so, either by launching Finder directly or by restoring the contents of FinderName to "Finder."

But, there is an exception to this scheme. The new Finder (4.1) provides an optional Finder replacement of its own, MiniFinder. Apple needed a mechanism to run MiniFinder if it is installed, or Finder if MiniFinder is not installed. What they did was to include an INIT resource (ID 4) in the system file.

An INIT resource is executed once, at system startup. INIT 4 installs a RAM patch to the OpenResFile (open resource file) trap. The patch code looks at the address of the filename that is being passed to OpenResFile. If that address is equal to FinderName, and if MiniFinder is present on the system disk, then the patch replaces the filename address with the address of the name "MiniFinder." The effect is that if MiniFinder is installed, the contents of FinderName is ignored. Note that Launch must do an OpenResFile on the application to be launched.

The net effect of this is that the presence of MiniFinder completely overrides FinderName, and there is no way to replace MiniFinder with an alternate except to remove MiniFinder.

You might have noticed also that when Finder launches an application on a system disk that is not the *current* system disk, the application disk will become the new system disk. But this is not true of MiniFinder: the disk from which MiniFinder was originally launched remains the system disk even

if the application disk has a system on it.

The reason for this is that Finder takes action, before launching the application, to switch systems; but MiniFinder does not do this. Before the launch, Finder checks to see if "System" and "Finder" exist on the application disk (if it's not the current system disk). If they do exist, then Finder closes the current system file (CloseResFile with an argument of 0), changes the contents of global variable BootDrive to reflect the new system disk, and does an InitResources to establish the new system file.

Low-level I/O

This and the following questions are paraphrased from exchanges in which I was involved in the Mac Developer's Forum on CompuServe. This question was occasioned by the sixth paragraph of "Writing Your Own Device Drivers" on p. 25 of the Device Manager section of *Inside Macintosh*.

Q. *Inside Macintosh* implies that a device driver must handle asynchronous or immediate requests differently from normal ones. How can the driver determine whether those attributes apply to the I/O request?

A. In fact, the driver does *not* need to know whether its current request is asynchronous and/or immediate. (But if it did want to know, it could examine the ioTrap field of the parameter block).

First, a brief review of some terminology. An *asynchronous* I/O request is one that will allow control to be returned to the requesting application before the request has been completed. Compare with a *synchronous* request, which is completely processed before the I/O Manager returns from the trap which issued the request. An asynchronous request is made by setting bit 10 of the trap word; in assembly language this is done by coding ",ASYNC" after the trap macro.

An *immediate* request is one that will not be put on the device driver's queue of pending requests; instead, it

will be given to the driver immediately, ahead of any normal requests that may be in its queue. An immediate request is made by setting bit 9 of the trap word; in assembly language this is done by coding "IMMED" after the trap macro.

There is an ambiguity in the two uses of the word "asynchronous" at the bottom of p. 25 of the Device Manager section. As I read it, an "asynchronous call" is the result of a trap with the ASYNC bit set; on the other hand, "asynchronous portions of the [driver] routines" are those portions which execute at interrupt level. Hence at a prime, control, or status entry point, all registers are available regardless of whether the trap is ASYNC (assuming that the entry point is not shared by interrupt handling code); conversely, interrupt handling code must preserve registers D4-D7/A4-A7 regardless of what kind of trap initiated the I/O.

On p. 26, *IM* is wrong in saying that IMMED-invoked routines *must* return via RTS rather than via IODone. IODone looks at the IMMED bit in the trap word and, if it's set, does not dequeue the request (because it's not on the queue) nor does it check to see if the driver has more work waiting in the queue. If IMMED is set, all IODone does is unlock the driver (make it relocateable) provided that it's not in ROM and its dNeedLock bit is clear.

In short, the driver needn't be concerned with whether the I/O request is asynchronous or immediate; the I/O OS routines those concerns.

Relocateable Code

Q. If I have a data area in my program, such as

```
Data: DC.B 'This is some
      data'
```

how can I force the MDS assembler to let me use the instruction `Move.L #Data,(A1)`? It issues an error message. But I don't want the extra overhead of

```
Lea Data,A0
Move.L A0,(A1)
```

A. MDS won't assemble absolute

(non-relocateable) code. For

```
Move.L #Data,(A1)
```

to work, Data would have to be at an absolute address known at assembly time or at link time. But the address of a code segment is not known until the segment is loaded at execution time: that's a characteristic of the Mac system architecture (not merely of MDS).

If it's any consolation,

```
Lea Data,A0
Move.L A0,(A1)
```

takes the same amount of memory and the same execution time as what you wanted to do -- the only cost is an additional line of source code.

Length of Relocateable Block

Q. I'm adding to a resource fork of a file using `AddResource`. How do I tell the Resource Manager the length of the new resource's data?

A. The length is that of the data to which the handle refers, i.e., what would be returned by `GetHandleSize`. The length of data referred to by a handle is determinable from the contents of Memory Manager data structures. If you're interested in how the Memory Manager keeps track of block lengths, see "Structure of Blocks" starting on p. 20 of the Memory Manager section of *Inside Macintosh*.

Replacing a File

Q. I'm using `SFPutFile` in the process of creating a file. If a file of the same name as specified by the user already exists, how do I know? I don't see anything in the `sfReply` record that would tell me. What am I missing?

A. You're not missing anything. `SFPutFile` will ask the user to confirm his choice of name if the filename is already in use on the volume, but it doesn't tell your program of that fact. There are several approaches you can use, all with basically the same effect, such as:

(1) Create the file. If you get a

`dupFNErr` code (file already exists), issue a `SetEOF` call to set the logical end-of-file point to zero.

(2) Delete the file. If you get a `fnfErr` code (file not found), ignore it. Then create the file.

(3) Issue a `GetFileInfo` request. If you get a `fnfErr` code, you know the file doesn't pre-exist.

Putting Screen to Sleep

Darkening the screen while the Mac is powered up but not in use is a good idea; it prevents the screen image from "burning" into the phosphor coating.

Q. I'm writing a "screensaver" routine to black out the screen after a period of inactivity. After saving the current `grafPort` and opening my own, I set the `bkPat` to black and do an `EraseRect` on the `portRect`. That works ok to blank out the screen; but restoring the previous `grafPort` doesn't restore the previous screen image. How can I restore it?

A. To get the old display back again, you have to use Window Manager facilities to update the desktop and windows, and `DrawMenuBar` to refresh the menu bar. Listing 2 shows an MDS Assembler routine that puts the screen "to sleep" with a randomly moving Apple symbol until the mouse is clicked or a key is pressed.

QuickDraw Globals

Q. I'm having trouble locating the QuickDraw global variables in assembler. Where are they?

A. Assuming that your program has initialized QuickDraw in the conventional manner, i.e.,

```
Pea -4(A5)
_InitGraf
```

then A5 contains a pointer to the address of the "first" field of the QuickDraw global area. Another way to think of it is that A5 is a "handle" to the QuickDraw globals in the sense of a pointer to a pointer (but not in the strict Memory Manager sense).

After

Move.L (A5),A0

A0 contains the address of the "first" field in the QuickDraw global area. I put "first" in quotes because it's the first field (namely, thePort) as listed in *Inside Macintosh*, QuickDraw section p. 34; but with respect to order in memory, it's last (highest address). In other words, after executing the above instruction, the following would hold:

Move.L (A0),A1 ;A1 contains address of
 ;the current grafPort
Lea -8(A0),A1 ;A1 contains address of
 ;white pattern
Lea -16(A0),A1 ;A1 contains address of
 ;black pattern
...etc.

Listing 1

```
; SCREEN_SLEEP
;© by Steve Brecher for MacTutor 1985
;
; Darkens the screen except for a randomly-moving Apple
;symbol. Returns when the mouse button is pushed or when a
;key is pressed. The mouse click or keypress event is
;removed from the event queue.
;
; D0/D1/D2/A0/A1 are destroyed; other registers are
;preserved.
;
    Include MDS:MacTraps.D
    Include MDS:SysEqu.D
    Include MDS:ToolEqu.D
    Include MDS:QuickEqu.D

XDEF Sleep

mDownMask    Equ    2      ;Equate files missing these...
keyDownMask  Equ    8

Macro    Pop Dest =
Move    (SP)+,{Dest}
|
Macro    Pop.L Dest =
Move.L  (SP)+,{Dest}
|
Macro    Push Src =
Move    {Src},-(SP)
|
Macro    Push.L Src =
Move.L  {Src},-(SP)
|

Sleep
    Push.L    A2          ;save
    _HideCursor          ;sleepy screens show no
                          ;cursors
```

```
SubQ      #4,SP          ;space to store old grafPort
                          ;address
Push.L     SP            ;pass address of space to
                          ;_GetPort
_GetPort   ;get the old grafPort addr
MoveQ      #portRec,D0   ;size of a grafPort record
_NewHandle ;allocate on heap
_Hlock     ;bolt it down
Move.L     (A0),A2       ;get pointer from handle
Push.L     A0            ;save handle for later
Push.L     A2
_OpenPort  ;default values to grafPort,
          ;make it current

Pea        portRect(A2)
_PaintRect ;paint with default fillPat =
          ;black
Bsr.S      RandApple     ;move an Apple around 'til
          ;click or key
Pop.L      A0            ;handle to our grafPort
_DisposHandle ;bye, grafPort
_DrawMenuBar ;restore the menu bar
Clr.L      -(SP)         ;indicate desktop to PaintOne
Push.L     GrayRgn       ;the region to be painted =
          ;whole desktop
_PaintOne  ;repaint the desktop
SubQ      #4,SP          ;space for FrontWindow result
_FrontWindow ;get parameter for
          ;PaintBehind
Push.L     GrayRgn       ;clobbered region = whole
          ;desktop
_PaintBehind ;repaint windows (if any)
_ShowCursor ;wake up cursor
_SetPort   ;restore previous grafPort
          ;from stack
Pop.L      A2            ;restore register
```

Rts

; Bop an Apple symbol randomly about the screen, moving it
; once a second. A2 is presumed to have the address of the
; current grafPort, with full-screen bounds. Note that the
; Apple's bottom will rest on the character baseline (where
; _MoveTo puts the pen) and its left edge will be at the pen
; position. To avoid clipped Apples we keep the pen position at
; least an "Apple-height" below the top and at least an
; "Apple-width" from the right edge. We get the width with
; _Charwidth, and assume height=width.

RandApple

```
MoveM.L   D3/D4,-(SP)
Link      A6,#-evtBlkSize ;allocate space for local
                          ;EventRcd
Move      #srcXor,txMode(A2) ;drawing a char will invert
Clr.L     -(SP)           ;word for width, with 0 word
                          ;underneath
Push      #appleMark     ;char code for bitten apple
_CharWidth ;(SP) hi word = Apple width (&
          ;approx height)

@0SubQ    #2,SP
_Random
MoveQ     #0,D3           ;to get upper word of D3 clear
Pop       D3             ;D3.L has a random 16-bit
```

```

;value
Move    portRect+bottom(A2),D0 ;height of screen
Sub     (SP),D0                ;less approx height of Apple
                                ;(it's sorta square)
Divu    D0,D3                  ;divide by height of allowable
                                ;pen range
Add.L   (SP),D3                ;add Apple height to
                                ;remainder in high word
SubQ    #2,SP                  ;now similar logic for
                                ;horizontal coordinate...

_Random
MoveQ   #0,D1
Pop     D1
Move    portRect+right(A2),D0
Sub     (SP),D0                ;keep pen at least
                                ;Apple-width from right edge

Divu    D0,D1
Swap    D1                     ;get remainder in low word
Move    D1,D3                  ;D3 hi = vertical, D3 lo =
                                ;horizontal
Push.L   D3                    ;pass coordinates to
                                ;MoveTo...
_MoveTo
                                ;position pen to draw the
                                ;apple
MoveQ   #60,D4                 ;one second worth of ticks...
Add.L   Ticks,D4               ;+ current time, get ending
                                ;time
Push     #appleMark
_DrawChar
                                ;draw the apple (note: moves
                                ;pen)
@1Lea    -evtBlkSize(A6),A0
MoveQ   #mDownMask!keyDownMask,D0
_GetOSEvent
Beq.S   @2                     ;mouse click or keypress?
Cmp.L   Ticks,D4               ;branch if so, our cue to exit
Bhi.S   @1                     ;time to move the apple?
Bhi.S   @1                     ;branch if not
Push.L   D3                    ;yes, reposition (do a
                                ;"backspace")...
_MoveTo
                                ;...back to where the apple
                                ;was drawn
Push     #appleMark
_DrawChar
                                ;redraw to erase (srcXor)
Bra.S    @0                    ;go find another spot for the
                                ;apple
@2Unlk    A6
MoveM.L  (SP)+,D3/D4
Rts

```

Listing 2

```

REM Set up test data for Sort routine; do a timed call
REM and verify that routine worked.
DIM Test$(599) ' 600 strings
TotLen = 0
FOR I% = 0 TO 599
    Test$(I%) = STR$(RND)
    TotLen = TotLen + LEN(Test$(I%))
NEXT
PRINT USING "Average length of the 600 strings
is ##.##";TotLen/600
PRINT "Sorting..."

```

```

Start = TIMER
CALL Sort(Test$())
PRINT "Sort took";TIMER-Start;"seconds; checking
correctness..."
FOR I% = 0 TO 598
    IF TEST$(I%) > TEST$(I%+1) THEN PRINT "Oops, didn't
work!" : END
NEXT
PRINT "It worked!"
END

REM -----Sort Subprogram follows-----
REM
SUB Sort(S$(1)) STATIC
REM
REM Optimized Quicksort subprogram to sort
REM an array of strings into ascending order.
REM Algorithm adapted from:
REM Sedgewick, Com. of the ACM, V21 N10, Oct. 1978
REM and corrigendum, V22 N6, Jun. 1979
REM Also see Chapter 9 of Sedgewick, "Algorithms",
REM Addison-Wesley, 1983, ISBN 0-201-6672-6
REM
REM Rather than use recursion, use a stack of array
REM partitions --
REM The "Dimmed" kludge seems to be required to get
REM a truly local array
REM in a SUB that may be called more than once:
    IF NOT Dimmed THEN DIM Stack%(15,2)
        ' 15 handles 32,768 elements
    Dimmed = 1
REM
REM Sedgewick's "M" parameter, which determines when to
REM stop Quicksorting and
REM finish up with an insertion sort on entire
REM array (an optimization):
    Insertion% = 10
REM
    L% = LBOUND(S$) ' "left" subscript
    R% = UBOUND(S$) ' "right" subscript
    IF L% = R% THEN EXIT SUB ' One element is easy to sort!
    IF R% - L% < Insertion% THEN GOTO InsertionSort
    StackPtr% = 0
REM Initialize for partitioning the subarray such
REM that the partitioning
REM element S$(L%) is the median of: old S$(L%),
REM S$(Middle%), S$(R%)
PartInit:
    Middle% = (L%+R%) / 2
REM Lines beginning with "Temp$ =" are exchanges of
REM array elements
    Temp$ = S$(Middle%) : S$(Middle%) = S$(L%)
    S$(L%) = Temp$
    IF S$(L%+1) <= S$(R%) THEN GOTO P2
    Temp$ = S$(L%+1) : S$(L%+1) = S$(R%) : S$(R%) = Temp$
P2:
    IF S$(L%) <= S$(R%) THEN GOTO P3
    Temp$ = S$(L%) : S$(L%) = S$(R%) : S$(R%) = Temp$
P3:
    IF S$(L%+1) <= S$(L%) THEN GOTO Partition
    Temp$ = S$(L%+1) : S$(L%+1) = S$(L%) : S$(L%) = Temp$
Partition:
    I% = L%+1

```

```

J% = R%
Partitioner$ = S$(L%)
Incl:
  I% = I% + 1
  IF Partitioner$ >= S$(I%) THEN GOTO Incl
DecJ:
  J% = J% - 1
  IF S$(J%) > Partitioner$ THEN GOTO DecJ
  IF I% >= J% THEN GOTO GotIJ
  Temp$ = S$(I%) : S$(I%) = S$(J%) : S$(J%) = Temp$
  GOTO Incl
GotIJ:
  S$(L%) = S$(J%)
  S$(J%) = Partitioner$
REM Determine what to do next depending on relative and
REM absolute sizes of subarrays
NL% = J% - L%      ' size of left subarray
NRM1% = R% - I%    ' (size of right subarray) - 1
IF NL% > NRM1% THEN GOTO BigLeft
REM Right subarray is larger
IF NRM1% < Insertion% THEN GOTO CheckStack
IF NL% > Insertion% THEN GOTO LeftNext
REM Partition right subarray next
L% = I%
GOTO PartInit
REM Left subarray is larger (or equal)
BigLeft:
  IF NL% <= Insertion% THEN GOTO CheckStack
  IF NRM1% >= Insertion% THEN GOTO RightNext
  R% = J% - 1
  GOTO PartInit
REM "Push" right subarray, partition left subarray next
LeftNext:
  StackPtr% = StackPtr% + 1
  Stack%(StackPtr%,1) = I%

```

```

Stack%(StackPtr%,2) = R%
R% = J% - 1
GOTO PartInit
REM "Push" left subarray, partition right subarray next
RightNext:
  StackPtr% = StackPtr% + 1
  Stack%(StackPtr%,1) = L%
  Stack%(StackPtr%,2) = J% - 1
  L% = I%
  GOTO PartInit
REM If stack not empty, pop and partition; else finish
REM with insertion sort
CheckStack:
  IF StackPtr% = 0 GOTO InsertionSort
REM Pop subarray specifier stack into L%, R%
L% = Stack%(StackPtr%, 1)
R% = Stack%(StackPtr%, 2)
StackPtr% = StackPtr% - 1
GOTO PartInit
REM
REM Insertion sort on entire array
REM
InsertionSort:
  FOR I% = UBOUND(S$)-1 TO LBOUND(S$) STEP -1
    IF S$(I%+1) > S$(I%) THEN GOTO Loop
    Work$ = S$(I%)
    J% = I% + 1
Slide: S$(J%-1) = S$(J%)
    J% = J% + 1
    IF J% <= UBOUND(S$) THEN IF Work$ >= S$(J%)
      THEN GOTO Slide
    S$(J%-1) = Work$
Loop: NEXT
  END SUB
REM-----End of Sort subprogram-----

```



Assembly Language Lab

Grow Window in MacAsm



Show Windows

Jan Eugenides
MacTutor Vol. 1 No. 11

There are several stages a programmer goes through when learning to program the Macintosh. The first is to learn about the tools available to you, how they work and what they can do. The second is to unlearn everything you know about interacting with the person using the program. This in itself is quite an undertaking. Then and only then can you begin putting the tools together into a complete program that works like a Macintosh program should work. This article, and the program which accompanies it, is designed to help you with the first stage - learning to use the Macintosh Tool Kit.

This article deals with assembly language programming, to fill the need for technical information directly related to Macintosh programming. Apple Computer, Inc. is not overly supportive of assembly language programming, and most of Inside Macintosh deals with Pascal examples and procedures. I'm not knocking Apple, they had to choose some language to use in the examples, and I suppose Pascal was a logical choice. They did include some very helpful information for assembly language programmers at the end of each chapter. However, the assembly language programmer is still faced with translating the Pascal procedures and definitions into the necessary stack setups and such. Not an easy task, especially if you are not familiar with Pascal's inner workings.

What It Does

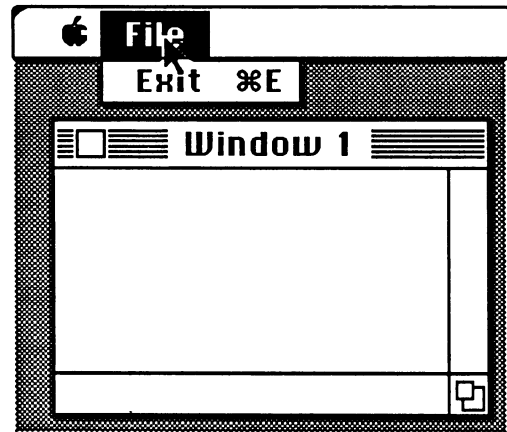
This month's program is a complete, double-clickable application, designed to demonstrate some very useful programming techniques. You can use it as a "shell" for writing your own applications. Everything is in place and working, all you have to do is insert your own code in the appropriate places.

The program, called "Show Windows," displays three common types of windows, and shows how to handle each one properly from assembly language. It also shows how to set up and display the menu bar, how to get events from the Mac operating system (in this case mouse events), and shows how to create resource files such as icons and dialog boxes for use by the application. Desk accessories are fully supported, and the application can be run by double-clicking its icon.

Running the Program

Before you run the application you must assemble and compile it (see the "Typing in the Program" section of this article for more details about assembling and compiling).

When you double-click the icon, you'll see the familiar zoom boxes, the menu bar will be redrawn, and a window with the title "Window 1" will appear. You can drag and re-size the



window with the mouse as usual. Clicking in the goaway box will close the window, and a new one will appear. Three types of windows can be seen in this fashion.

When the program first runs, you'll notice the disk whirrs when you close a window, before the next one is drawn. It also whirrs the first time you open the dialog box. This is the resources being fetched from the disk. After this, they will be in memory and everything will run much faster the next time you display the windows or dialog box. They will stay in memory until another application or accessory needs the space and then they'll be purged. When you need them again after that, they'll have to be re-loaded from the disk.

Open a desk accessory or two, and experiment with clicking on the various windows, moving them about. As you can see, everything runs concurrently with no problems.

Typing in the Program

This application was written using the Mainstay MasASM assembler. The MacASM assembler is a very fast 68000 assembler. It does not use the Mac interface much, but is more like running an assembler on a conventional computer. In fact, it is very similar to the S-C Macro Assembler for the Apple II. I admit I would like to see it use the mouse for editing, but otherwise it's great, and it is FAST.

MacASM comes with several examples on the disk, written by Yves Lempereur. They certainly helped me get this application working, thanks, Yves!

MacASM also comes with an automatic resource compiler that will translate resource definitions into actual resource files. This makes life easy for the programmer. Ordinarily, I just use "filler" resources, standard templates which are assembled with the program. Then, I use ResEdit

to change them into the actual resources I need. That's generally a faster and easier method. But for the sake of demonstration, I have spelled out all the resources as they should be in this example.

After you have typed in the source code and assembled it, all you need to do is double-click the assembled program's icon, and the compiler will automatically run and compile the assembled binary file into a Macintosh application file with a data and resource fork. The application is then ready to run.

Blow by Blow Description

From here on out I'll be dealing with the technical description of the program. I'm assuming you are already familiar with 68000 assembly language, and with the information in "Inside Macintosh."

Line 250 starts off the program with an `.INCLUDE "Library.Asm"` directive, which causes the file "Library.Asm" to be included in the assembly. This is a file which is included on the MacASM disk, and contains all the trap words for the various Mac ROM calls, as well as some simple macro definitions, such as `DEFV` which simulates the "define variable" directive, and `GLOBAL` which handles allocation of global variable space. `TBX` is the macro for trap calls. [MDS users can use similar library files for the trap macro names. -Ed]

Lines 270-420 are setting up storage for the variables used in the program, and by the system. Line 460 sets the target file to `SW.BIN` [this is specific to MacAsm. -Ed.], and line 480 sets the resource file to `Show Window 1.0 - 05/05/85`.

The Resources

Next come the resource definitions. Resources are data which the program can call from disk or memory for its use, and are separate from the actual code. That makes it easy to change them later, without reassembling or recompiling the program.

Lines 500-520 are the "owner" resource, an application-defined file. Lines 540-700 are the application's icon, [See last month's article by Steve Rabalais on creating an icon for MacAsm with his version of the Icon Converter. -Ed.] and lines 720-870 are the icon mask, which will affect how the icon looks when highlighted, and when it's moved onto the desktop. Next, in lines 900-1060 are standard File reference and Bundle resources.

Lines 1080-1200 are the Menu 1 resource. This data will be added to the first menu (the Apple menu) by the program when it runs. Lines 1220-1320 are the Menu 2 resource, and will comprise the second menu when the program runs.

Lines 1340-1420 contain a dialog box definition, with the size, location, type, and other parameters. Following that in lines 1440-1640 are the items which will go in the dialog box, including an "OK" button and some text.

Lines 1660-1910 contain the window templates for the three windows the program can display. These are standard

type 0, type 4, and type 16 windows, as described in Inside Macintosh.

And that concludes the resource definitions. At assembly time, they are assembled as hex data, and after assembly, the compiler converts the whole shebang into a complete application with a data and resource fork. The process is quite easy and fast.

The Code

Lines 1930-3960 comprise the actual application code. All the various trap calls are commented, so I won't go into them here, but you should notice the way the stack is set up prior to each call. All the ROM calls expect the stack to be set up like it would be set up by Pascal. When programming in assembly, it's up to you to properly initialize the stack before each call, and then to remove any data which is returned there. It takes a little getting used to, but once you know the setups for the various calls, it's not too bad. [Note: back issues of MacTutor have covered events and windows in some detail, so you may wish to refer to these for more specific information on how windows are constructed and how the event manager works. See especially numbers 3 and 4 -Ed.]

Lines 1940-2260 handle the initialization of the various necessary routines, inserting the applications menu items and redrawing the menu bar, and drawing the first window.

Event Driven

The application is entirely event driven, which means that it simply loops until an event is detected by the system and passed on to the application. Events are such things as clicking the mouse button, pressing a key on the keyboard, inserting a disk, etc. The application can either respond to the event, or ignore it. This application ignores all events except `Mousedown`, `Update`, and `Activate` events. (`Mousedown` is clicking, `Update` means a window must be redrawn, and `Activate` means a window is becoming active.)

Line 2280 begins the loop with a call to `SystemTask`, which allows the system to do whatever it wants while the application is idling. This is necessary to allow desk accessories to operate. Then the next event is fetched, and depending on what it is, the program jumps to the proper subroutine. If an event is to be ignored, the program just returns to the loop.

Lines 2520-2570 take care of an `Activate` event. When a new window is activated, all that's necessary in this application is to test to see if it is window #1. If it is, a grow box is drawn. The other windows do not have grow boxes.

Lines 2580-2720 handle an `Update` event. The grafport is set to the current port, the background is replaced, and again the grow box is drawn if necessary.

Lines 2760-3960 handle the various types of `Mousedown` events. First, lines 2760-2900 figure out where the mouse button was pressed. It can be in the menu, on the desktop, in the content portion of a window, in the drag bar of a window, in the grow box, or in the go away box.

Lines 2910-3360 handle menu selections. Lines 2910-3020 determine which menu and highlight it, then determine which item of the menu and branch accordingly.

Lines 3040-3160 handle the items in Menu 1. If it's the first item, it's our "About Show Windows..." selection, and a branch is made to lines 3180-3310 which display the dialog box. If it is not the first item, the parms are passed to OpenDeskAcc which opens the proper desk accessory.

Lines 3180-3310 open our dialog box, make it modal so it can't be dismissed, and wait for the OK button to be pressed. Then, the dialog box is closed and the loop continues.

Lines 3330-3360 handles Menu 2 selections. If item 1 is selected, the program ends with an ExitToShell.

Lines 3370-3400 handle a mousedown on the desktop with a call to SystemClick, which would allow any accessories to handle the event as they chose. With the Mac, you must be polite because you don't know who you might be working with. [Note that desk accessories that support TextEdit are not operable since they require an Edit menu, which we have not added. Hence if you try to type into the note pad, the program will crash. As an exercise, try to add support for typing into the note pad. If you get stuck, see MacTutor number 4 for a complete support of the note pad in MDS assembly.-Ed.]

Lines 3410-3480 take care of clicking in the content portion of a window. If it's our window, a call to SelectWindow will activate it.

Lines 3490-3520 handle dragging the windows around the desktop.

Lines 3540-3690 change the size of the window, first drawing the grey outline with a call to GrowWindow, and then resizing the window with a call to SizeWindow. Then the grafport is reset, and the window manager is notified that the window has changed. This will result in the proper update and activate events being generated.

Lines 3700-3960 handle closing windows. After a window is closed, the next type of window is drawn, over and over, so that closing one window automatically creates another of the next type.

Finally some boundaries for the drag are defined, and the necessary segment zero definition ends the program.

Possible Uses

The source code for this program is purposely written in such a way that you can easily modify it for your own uses. The jump tables are all complete, containing all the various events even though they are not all used in this program. All you need to do is insert your own code in the right places to produce a working application of your own.

```
00010 *SAVE S.SHOW.WINDOW
```

```
00020 *-----
```

```
00030 ; This is a simple stand alone
```

```
00040 ; double-clickable application.
```

```
00050 ; It has its own icon and other
```

```
00060 ; resources. It is intended as an
```

```
00070 ; example of how to create an
```

```
00080 ; application from assembly language.
```

```
00090 ; All it does is create three common
```

```
00100 ; types of windows. Closing one window
```

```
00110 ; causes the next type to appear.
```

```
00120 *-----
```

```
00130 ; by Jan Eugenides
```

```
00140 ; for MacTutor, Oct. 1985
```

```
00150 ; 11601 N.W 18th St.
```

```
00160 ; Pembroke Pines, FL 33026
```

```
00170 ;
```

```
00180 ; Many thanks to Yves Lempeurer for
```

```
00190 ; his examples and help.
```

```
00200 ;
```

```
00210 ; MacASM assembler
```

```
00220 *-----
```

```
00230 list off
```

```
00240 *-----
```

```
00250 INCLUDE "Library.Asm"
```

```
00260 *-----
```

```
00270 GLOBAL 52,$200
```

```
00280 ;
```

```
00290 DEFV W,ITMHIT
```

```
00300 DEFV L,WINDOW1
```

```
00310 DEFV L,WINDOW2
```

```
00320 DEFV L,WINDOW
```

```
00330 DEFV W,MENU
```

```
00340 DEFV W,MENUITM
```

```
00350 DEFV 16,DSKNAM
```

```
00360 DEFV 0,EVENTRECORD
```

```
00370 DEFV W,WHAT
```

```
00380 DEFV L,MESSAGE
```

```
00390 DEFV L,WHEN
```

```
00400 DEFV L,WHERE
```

```
00410 DEFV W,MODIFY
```

```
00420 DEFV W,CURWINDOW
```

```
00430 ;
```

```
00440 ENDG
```

```
00450 *-----
```

```
00460 TFILE "SW.Bin"
```

```
00470 *-----
```

```
00480 RFILE "Show Windows", APPL,JWIN,$2000
```

```
00490 *-----
```

```
00500 RSRC JWIN,0
```

```
00510 STR "Show Window 1.0 -- 05/05/85"
```

```
00520 ENDR
```

```
00530 *-----
```

```
00540 RSRC ICN#,128
```

```
00550 HEX 00000100FFFFFFFF Icon
```

```
00560 HEX 8000000180000001
```

```
00570 HEX FFFFFFFF82000061
```

```
00580 HEX C2000051C200006D
```

```
00590 HEX E2000075B2000055
```

```
00600 HEX 9E00007F8600002F
```

```
00610 HEX 82000027BA000027
```

```
00620 HEX C60000278C000021
```

```
00630 HEX 88000029AFFFFFFF9
```

```
00640 HEX B601001983FFFFE9
```

```
00650 HEX 8A01002FE601004F
```

```
00660 HEX EC01004DEC01004F
```

```
00670 HEX B4010045AC010065
```

```
00680 HEX 96010031CA010011
```

```
00690 HEX E601001986010009
```

00700	HEX	8C01000DDFFFFFFF	
00710	*		
00720	HEX	FFFFFFFFFFFFFFFF	mask
00730	HEX	FFFFFFFFFFFFFFFF	
00740	HEX	FFFFFFFFFFFFFFFF	
00750	HEX	FFFFFFFFFFFFFFFF	
00760	HEX	FFFFFFFFFFFFFFFF	
00770	HEX	FFFFFFFFFFFFFFFF	
00780	HEX	FFFFFFFFFFFFFFFF	
00790	HEX	FFFFFFFFFFFFFFFF	
00800	HEX	FFFFFFFFFFFFFFFF	
00810	HEX	FFFFFFFFFFFFFFFF	
00820	HEX	F801807FFE01803F	
00830	HEX	FC01803FFC01807F	
00840	HEX	FC01807FFC01803F	
00850	HEX	FC01803FFC01801F	
00860	HEX	FC01800FFC01800F	
00870	HEX	FFFFFFFFFFFFFFFF	
00880	ENDR		
00890	*		
00900	RSRC	FREF,128	File ref
00910	ASC	"APPL"	
00920	DATA	/0	
00930	STR	""	
00940	ENDR		
00950	*		
00960	RSRC	BNDL,128	Bundle
00970	ASC	"JWIN"	
00980	DATA	/0	
00990	DATA	/2-1	
01000	ASC	"ICN#"	
01010	DATA	/1-1	
01020	DATA	/0,/128	
01030	ASC	"FREF"	
01040	DATA	/1-1	
01050	DATA	/0,/128	
01060	ENDR		
01070	*		
01080	RSRC	MENU,1	Menu 1
01090	DATA	/1	ID
01100	DATA	/0,/0	
01110	DATA	/0	
01120	DATA	/0	
01130	DATA	\$FFFFFFB	enable bits
01140	HEX	0114	Title
01150	STR	"About Show Windows..."	
01160	DATA	#0,#0,#0,#0	style...
01170	STR	"-"	
01180	DATA	#0,#0,#0,#0	
01190	STR	""	
01200	ENDR		
01210	*		
01220	RSRC	MENU,2	Menu 2
01230	DATA	/2	
01240	DATA	/0,/0	
01250	DATA	/0	
01260	DATA	/0	
01270	DATA	\$FFFFFFF	
01280	STR	"File"	
01290	STR	"Exit"	
01300	DATA	#0,#E,#0,#0	
01310	STR	""	
01320	ENDR		
01330	*		
01340	RSRC	DLOG,1	dialog res
01350	DATA	/50,/60,/180,/320	Bounds
01360	DATA	/1	Wind ID
01370	DATA	#1,#0	Visible
01380	DATA	#0,#0	NoGoAway
01390	DATA	0	refCon
01400	DATA	/1	Item ID
01410	STR	""	Title
01420	ENDR		
01430	*		
01440	RSRC	DITL,1	
01450	DATA	/2-1	Number of Items
01460	DATA	0	
01470	DATA	/90,/220,/120,/250	Coords
01480	DATA	#4	Item type length
01490	STR	"OK"	Content
01500	DATA	0	
01510	DATA	/2,/2,/118,/240	
01520	HEX	0892 static text	144 Bytes
01530	ASC	" Jan Eugenides"	
01540	HEX	0D	
01550	ASC	" Assembly Corner"	
01560	HEX	0D	
01570	ASC	" 11601 N.W. 18th Street"	
01580	HEX	0D	
01590	ASC	" Pembroke Pines, FL 33026"	
01600	HEX	0D0D	
01610	ASC	"A simple application example"	
01620	HEX	0D	
01630	ASC	"from assembly language. "	
01640	ENDR		
01650	*		
01660	RSRC	WIND,128,32	Window
01670	DATA	/50,/150,/300,/380	Coord
01680	DATA	/0	Type 0
01690	DATA	#1,#0	visible
01700	DATA	#1,#0	GoAway
01710	DATA	0	refCon
01720	STR	"Window 1"	title
01730	ENDR		
01740	*		
01750	RSRC	WIND,129,32	
01760	DATA	/50,/150,/300,/380	
01770	DATA	/4	Type 4
01780	DATA	#1,#0	
01790	DATA	#1,#0	
01800	DATA	0	
01810	STR	"Window 2"	
01820	ENDR		
01830	*		
01840	RSRC	WIND,130,32	
01850	DATA	/50,/150,/300,/380	
01860	DATA	/16	Type 16
01870	DATA	#1,#0	
01880	DATA	#1,#0	
01890	DATA	0	
01900	STR	"Window 3"	
01910	ENDR		
01920	*		

```

01930 SEG 1,52 segment 1

01940 START PEA -4(A5)
01950 TBX InitGraf init QD
01960 TBX InitFonts font mgr
01970 TBX InitWindows window mgr
01980 TBX InitMenus menu mgr
01990 CLR.L -(SP)
02000 TBX InitDialogs dialog mgr
02010 CLR.L -(SP) menu handle
02020 MOVE.W #1,-(SP) meni id 1
02030 TBX GetRMenu get menu
02040 MOVE.L (SP),-(SP) push again
02050 CLR.W -(SP) append...
02060 TBX InsertMenu add menu item
02070 MOVE.L #$44525652,-(SP) 'DRVR'
02080 TBX AddResMenu Add desk Acc

02090 CLR.L -(SP) menu handle
02100 MOVE.W #2,-(SP) file menu
02110 TBX GetRMenu get menu
02120 CLR.W -(SP) append...
02130 TBX InsertMenu insert menu
02140 TBX DrawMenuBar menu bar

02150 MOVE.W #128,CURWINDOW(A5) rscs id
02160 CLR.L -(SP) window ptr. return
02170 MOVE.W CURWINDOW(A5),-(SP) id
02180 CLR.L -(SP) heap window
02190 MOVE.L #-1,-(SP) front window
02200 TBX GetNewWindow make window
02210 MOVE.L (SP)+,WINDOW1(A5) save ptr
02220 MOVE.L WINDOW1(A5),-(SP) push ptr
02230 TBX DrawGrowlcon add grow icon
02240 MOVE.L #$0000FFFF,D0 all events
02250 OST FlushEvents clear all
02260 TBX InitCursor show cursor
02270 *-----

02280 LOOP TBX SystemTask check DA
02290 CLR.W -(SP) returned event
02300 MOVE.W #-1,-(SP) mask all events
02310 PEA EVENTRECORD(A5)
02320 TBX GetNextEvent Get an event
02330 MOVE.B (SP)+,D0 result ret.
02340 BEQ.S LOOP nothing...

02350 MOVE.W WHAT(A5),D0 got one
02360 ADD.W D0,D0 table two bytes
02370 MOVE.W ETABLE(PC,D0.W),D0
02380 JMP ETABLE(PC,D0.W) go do it

02390 ETABLE DATA /LOOP-ETABLE
02400 DATA /MOUSEDWN-ETABLE
02410 DATA /MOUSEUP-ETABLE
02420 DATA /KEYDWN-ETABLE
02430 DATA /LOOP-ETABLE key up
02440 DATA /AUTOKEY-ETABLE
02450 DATA /UPDATE-ETABLE
02460 DATA /DISKINS-ETABLE
02470 DATA /ACTIVATE-ETABLE
02480 DATA /LOOP-ETABLE networking

```

```

02490 DATA /LOOP-ETABLE device driver
02500 DATA /LOOP-ETABLE app

02510 MOUSEUP BRA LOOP Mouse up, just loop

02520 ACTIVATE MOVE.W CURWINDOW(A5),D0
02530 CMP.W #128,D0 is it window 1?
02540 BNE.S .1 nope
02550 MOVE.L WINDOW1(A5),-(SP) yes
02560 TBX DrawGrowlcon
02570 .1 BRA LOOP and return to loop

02580 UPDATE MOVE.L MESSAGE(A5),-(SP)
02590 TBX BeginUpDate
02600 MOVE.L MESSAGE(A5),-(SP)
02610 TBX SetPort set graphport
02620 MOVE.L MESSAGE(A5),A0
02630 PEA 16(A0) rect in window rec
02640 TBX EraseRect wipe out

02650 MOVE.W CURWINDOW(A5),D0
02660 CMP.W #128,D0 is it window 1?
02670 BNE.S .1 no
02680 MOVE.L MESSAGE(A5),-(SP)
02690 TBX DrawGrowlcon yes
02700 .1 MOVE.L MESSAGE(A5),-(SP)
02710 TBX EndUpDate
02720 BRA LOOP return to loop

02730 AUTOKEY BRA LOOP

02740 KEYDWN BRA LOOP

02750 DISKINS BRA LOOP

02760 MOUSEDWN CLR.W -(SP) ret result
02770 MOVE.L WHERE(A5),-(SP) where?
02780 PEA WINDOW(A5) window ptr addr
02790 TBX FindWindow
02800 MOVE.W (SP)+,D0 get resykt
02810 ADD.W D0,D0 time 2...
02820 MOVE.W WTABLE(PC,D0.W),D0
02830 JMP WTABLE(PC,D0.W) do it

02840 WTABLE DATA /LOOP-WTABLE in desk...
02850 DATA /INMENU-WTABLE menu bar...
02860 DATA /SYSEVNT-WTABLE system...
02870 DATA /CONTENT-WTABLE content...
02880 DATA /DRAG-WTABLE drag area...
02890 DATA /GROW-WTABLE grow area...
02900 DATA /GOAWAY-WTABLE goaway...

02910 INMENU CLR.L -(SP) ret menu choice
02920 MOVE.L WHERE(A5),-(SP)
02930 TBX MenuSelect which menu?
02940 MOVE.L (SP)+,D7 menu id & item

02950 FUNCTION MOVE.L D7,MENU(A5) save menu
02960 CLR.W -(SP) select all menus
02970 TBX HiliteMenu unhilite all
02980 MOVE.W MENU(A5),D0 get menu id
02990 CMP.W #1,D0

```



```

03000      BEQ.S  INMENU1      its menu 1
03010      CMP.W  #2,D0
03020      BEQ.S  INMENU2      its menu 2
03030      BRA   LOOP          no one else

03040 INMENU1 MOVE.W MENUITM(A5),D0 apple menu
03050      CMP.W  #1,D0 menu item 1?
03060      BEQ.S  LOGO          about item...

03070      CLR.L  -(SP)         no, must be DA...
03080      MOVE.W #1,-(SP)
03090      TBX   GetRMenu       not item 1
03100      MOVE.W MENUITM(A5),-(SP)
03110      PEA   DSKNAM(A5) holder for DA
03120      TBX   GetItem       find DA name

03130      CLR.W  -(SP)         ret result
03140      PEA   DSKNAM(A5)
03150      TBX   OpenDeskAcc do DA...
03160      MOVE.W (SP)+,D0 ret result...
03170      BRA   LOOP          and loop

03180 LOGO  CLR.L  -(SP)       dialog wnd ptr
03190      MOVE.W #1,-(SP) dialog box id
03200      CLR.L  -(SP)       store on heap
03210      MOVE.L #-1,-(SP) in front...
03220      TBX   GetNewDialog get the box
03230      MOVE.L (SP)+,D7 save result
03240 .0    CLR.L  -(SP)       no filter proc
03250      PEA   ITMHIT(A5) VAR hit item
03260      TBX   ModalDialog it's modal
03270      MOVE.W ITMHIT(A5),D0 what happen
03280      CMP.W  #1,D0 time to exit?
03290      BNE.S  .0 no, wait for press
03300      MOVE.L D7,-(SP) get wind. ptr
03310      TBX   CloseDialog and close
03320      BRA   LOOP          return

03330 INMENU2 MOVE.W MENUITM(A5),D0 file menu
03340      CMP.W  #1,D0 item 1?
03350      BNE   LOOP          no, loop

03360      TBX   ExitToShell yes, exit

03370 SYSEVNT PEA  EVENTRECORD(A5)
03380      MOVE.L WINDOW(A5),-(SP)
03390      TBX   SystemClick handle it
03400      BRA   LOOP          and loop

03410 CONTENT MOVE.L WINDOW(A5),A0
03420      CMP.L  WINDOW1(A5),A0 our window?
03430      BEQ.S  .0 yes...

03440      CMP.L  WINDOW2(A5),A0 window2?
03450      BNE   LOOP          no, loop again
03460 .0    MOVE.L A0,-(SP)
03470      TBX   SelectWindow Select it
03480      BRA   LOOP          and loop

03490 DRAG  MOVE.L WINDOW(A5),-(SP) window ptr
03500      MOVE.L WHERE(A5),-(SP) where
03510      PEA   BOUNDS(PC) drag boundary

```

```

03520      TBX   DragWindow and drag it
03530      BRA   LOOP

03540 GROW  CLR.L  -(SP)       grow box event
03550      MOVE.L WINDOW(A5),-(SP) window ptr
03560      MOVE.L WHERE(A5),-(SP) where?
03570      PEA   SIZE(PC)
03580      TBX   GrowWindow draw grow area
03590      MOVE.L (SP)+,D7
03600      MOVE.L WINDOW(A5),-(SP)
03610      MOVE.L D7,-(SP)
03620      MOVE.B #1,-(SP)
03630      TBX   SizeWindow resize window
03640      MOVE.L WINDOW(A5),-(SP)
03650      TBX   SetPort set graphport
03660      MOVE.L WINDOW(A5),A0
03670      PEA   16(A0) rect of graphport
03680      TBX   InvalRect force update evt
03690      BRA   LOOP          and loop

03700 GOAWAY CLR.W  -(SP)
03710      MOVE.L WINDOW(A5),-(SP)
03720      MOVE.L WHERE(A5),-(SP)
03730      TBX   TrackGoAway
03740      BEQ   LOOP          goaway canceled
03750 ;
03760      MOVE.L WINDOW(A5),-(SP)
03770      TBX   DisposWindow close window
03780      ADDQ   #1,CURWINDOW(A5)
03790      MOVE.W CURWINDOW(A5),D0
03800      CMP.W  #131,D0 all three yet?
03810      BEQ.S  .1 yes, start over

03820      BSR   DRAW.NEW.WINDOW no
03830      BRA   LOOP          and loop

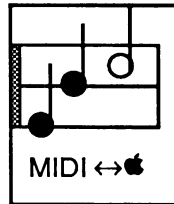
03840 .1      MOVE  #128,CURWINDOW(A5) reset
03850      BSR   DRAW.NEW.WINDOW draw it
03860      MOVE.L WINDOW1(A5),-(SP)
03870      TBX   DrawGrowIcon
03880      BRA   LOOP          and return to loop
03890 DRAW.NEW.WINDOW
03900      CLR.L  -(SP)
03910      MOVE.W CURWINDOW(A5),-(SP)
03920      CLR.L  -(SP)
03930      MOVE.L #-1,-(SP)
03940      TBX   GetNewWindow new window
03950      MOVE.L (SP)+,WINDOW1(A5)
03960      RTS   and return
03970 *-----
03980 BOUNDS DATA /24,/4,/338,/508
03990 SIZE DATA /32,/64,/290,/470
04000 ENDR
04010 *-----
04020 SEG 0,32,VAR.LEN,$20
04030 SEG0
04040 SEG_1 JP START,1
04050 END_1
04060 END0
04070 ENDR
04080 *-----
04090 END

```



Sound Lab

The Midi Connection



Kirk Austin
San Anselmo, CA.

This article will describe one way to interface a Macintosh with the MIDI standard for communication with music synthesizers and related equipment. By making a few hardware and software changes to the existing Macintosh serial ports, MIDI ports can be created. A Macintosh equipped with MIDI ports can act as a very powerful controller for musical equipment. By sending and receiving MIDI data the Mac can do things like record synthesizer music, edit it, play it back, and do an automated mixdown to stereo. The implications of this capability are great. Many of the functions of a modern recording studio can now be brought into the Macintosh owner's home.

MIDI stands for Musical Instrument Digital Interface and is a standard that was agreed upon by the various major manufacturers of electronic music synthesizers. The MIDI standard is a protocol for transferring data between music synthesizers and computers. It allows equipment made by different manufacturers to communicate with each other. Originally it was designed to let one keyboard instrument control another one. You could play the keyboard on synthesizer "A" and the sound would be produced by synthesizer "B". But the originators of the MIDI concept gave it enough flexibility to allow it to do other things as well. Now it is possible to buy MIDI gear that will control lighting equipment and mixing boards as well as keyboard synthesizers. An important point to remember is that MIDI recorders don't actually record the sound that is produced by the synthesizer but just the control signals that the synthesizer needs to produce a sound. For instance, when a key is depressed on a synthesizer keyboard at least three bytes of information are sent out over the MIDI cable: the code describing a note down event and MIDI channel, the number corresponding to the key (middle C=60), and the velocity that the key was struck with (used for dynamics). A note up event is recorded in a similar fashion. So, altogether it takes six bytes to record a note of any duration. This makes for a very compact data structure compared to actually recording the sound itself.

In order to use the MIDI ports on the Mac you will have to write your own serial drivers. As far as I know the driver routines for configuring the serial ports must be written in assembly language. If you have a high level language compiler, it should allow inline assembly code in order to take advantage of the information presented in this article. If your compiler doesn't allow this you may want to read on just to further your understanding (you may also want to consider getting another compiler). This month's coverage will be of the hardware interface necessary to convert the Mac's serial

ports to MIDI specifications. The software driver routines will be covered in a future article.

The most important consideration in converting to the MIDI way of doing things is the baud rate. MIDI is clocked at 31.25K which is a nonstandard rate. The Macintosh uses a Zilog 8530 chip to handle the serial ports. In order to get the Mac's 8530 chip to handle data at the MIDI clock rate, you have to provide an external clock signal for it to use as a timing reference. I have used an oscillator circuit and a divider in order to derive a 500K clock signal. This is again divided by 16 in the 8530 to produce the necessary clock rate of 31.25K. Note that you may optionally use other pins on the divider chip to get a frequency of 1 Meg or 2 Meg. Either of these rates will also do the trick if you select the appropriate divisor in the 8530's write register #4 (more on this in the forthcoming software description). Other than the external clock the interface consists of some level shifting and isolation circuitry in order to make the Mac's signals conform to the MIDI specification.

The circuit is pretty straightforward. First we need to generate the clock signal, which is done with the circuit built around the crystal. The two inverters, two resistors, capacitor and crystal form a 4 Meg oscillator that drives the 74LS93 binary divider chip. One of the outputs of the divider (depending on which frequency you want to use) is routed to a capacitor and then a resistor to ground that is used to offset the pullup resistors internal to the 26LS32 receiver chip in the Mac. The reason this is necessary is that the negative input of the 26LS32 is referenced to ground, so we need to swing at least 200 millivolts below ground in order for the 26LS32 chip to recognize it as a legitimate clock signal. This particular idiosyncrasy of the Mac is rather annoying, but not unworkable.

The transmitted data signal line is the next consideration. The TXD+ signal from pin 4 of the Mac is fed through a couple of inverter stages to provide the TTL output required by MIDI. The resistor and diode in front of the inverters protects them from the negative voltages produced by the Mac. This is the easiest part of the interface.

The MIDI input port uses an optoisolator to avoid a condition known as ground loops. This happens in audio systems when there is a slight voltage potential between the ground lines of two different pieces of equipment. The result being an irritating 60 cycle hum in the audio output. The optoisolator keeps the equipment from reacting this way by

The Electrical Mac

Macintosh PAL Technology

Jeff Mitchell
MacTutor Contributing Editor
MacTutor Vol.1 No. 11

Programmable Logic Devices

Among its other achievements, the Macintosh has stirred considerable interest and spawned many rumors on the subject of PALs®. Most computer enthusiasts are familiar with hardware terms like microprocessor, TTL, and UART. Programmable logic, of which PALs are just one of several types, is a relatively recent phenomenon, and hadn't received much attention from the general press until the Mac arrived with its logic board full of it. It seemed as if everyone was convinced Apple had been able to package some black magic in a mysterious device called a PAL.

My aim this month is to dispel the mystery surrounding PALs. I'll describe the various types of programmable logic devices (PLD), detail their various internal architectures, and explain what it is they actually do. I'll present an application which demonstrates the value of these devices, and finally I'll describe the functions the HAL devices in the Macintosh perform.

What's A PAL?

A PAL is one of several types of field programmable logic, of which the PROM is the best known. Field programmable means the user can program his logic into the device himself, rather than the logic being part of the semiconductor manufacturing process as with mask ROMs.

Current digital design practice consists mostly of tying several LSI devices together with some 'glue logic', consisting of some number of SSI and MSI TTL devices which serve to generate control signals for the LSI. Although necessary, this glue logic takes up a large amount of space due to its low level of integration. This is where programmable logic devices come in. They serve as replacements for random logic and provide much more functionality than SSI TTL devices in a similar size package. Signetics claims that the replacement of 15-20 I.C.'s with one 82S100 FPLA is not unusual. My experience is that replacement of 2-5 I.C.'s is a more realistic figure.

The reason programmable logic can be used to replace random glue logic is that any logic equation can be reduced to a sum of products; that is, groups of AND terms connected by OR terms. Not surprisingly, programmable logic devices consist of an array of AND gates feeding an OR gate.

Why Use PLDs?

There are many advantages to using programmable logic rather than discrete TTL devices. There is the PC board real estate that is saved when several devices are replaced by one, along with an accompanying power savings, although this can depend on how many devices were replaced. All but the newest PLDs are fabricated using bipolar technology, which consumes large amounts of power. On the other hand, bipolar technology is fast, and there is a lot less propagation delay through a single PLD than through several TTL devices, allowing the designer to increase circuit performance as well as save space.

The design stage of product development is where PLDs are most valuable. Changing the circuit logic is as simple as reprogramming a single chip with a different logic equation. This saves countless hours of rewiring, and lets the designer experiment with circuits he might otherwise not try.

PLDs play a role in manufacturing, too. A single device may replace several different types of TTL logic, reducing inventory and purchasing requirements. Bugs found after the product has been shipped may be correctable by changing the PLD, instead of costly cut-and-jumper rework to the PC board.

There's a down side to using programmable logic, though. These devices are significantly more expensive than jellybean TTL parts, on the order of several dollars as opposed to less than a dollar. But this is only the component price and does not include the savings from a smaller PC board, less assembly labor, less inventory, and perhaps a smaller power supply. There is also the option of turning a programmable device into a high-volume masked part, which is what Apple has done with the Macintosh.

PLDs are generally used where there is considerable random logic, or where space is at a premium, yet the production quantities cannot justify the expense involved in designing a fully custom device; i.e. a gate array or standard cell.

Types of PLDs

Bipolar proms were the first devices used to replace random logic, usually in address decoding applications. In 1975, Signetics Corporation developed the FPLA (Field Programmable Logic Array), which allowed a fully customizable implementation of sum-of-products equations. This was the first device aimed specifically at replacing random logic. Monolithic Memories, Inc. (MMI) soon followed with the PAL (Programmable Array Logic), which while similar to the FPLA in concept, differs in its internal implementation of the sum-of-products logic.

Both of these devices were based upon fuseable-link bipolar PROM technology, in which the device is programmed by injecting large currents into specific locations, vaporizing selected internal fuses. The remaining intact fuses determine what transfer equation will be implemented.

Since the introduction of PLDs, the technology has advanced, increasing the speed, programming yields, and reliability of the parts. The general architecture has not changed, although more complex devices have been introduced as the market matures and new applications appear.

PLD Notation and Logic Symbolism

PLDs have their own symbols and logic notation, which I'll introduce here, and then go on to describe the internal architecture of each of the three types of programmable logic, PROMs, PALs, and FPLAs.

An AND gate is normally portrayed as in Figure 1a. Figure 1b shows how this same gate is represented when implemented in programmable logic. The X's indicate an intact fuse. Figure 1c shows the device in (b) programmed, with input B removed from the equation. The lines still cross, but the missing X indicates that a fuse has been blown. Figure 1d is the same gate, but is not programmable. An intersection with a dot is hard-wired internally.

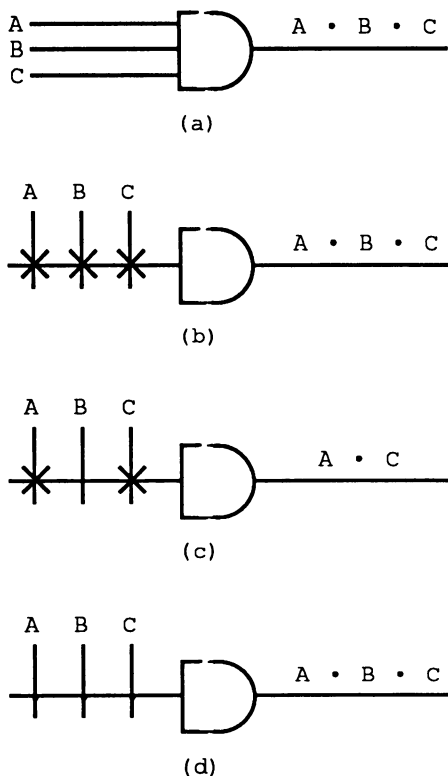


Figure 1 - PLD logic notation

Internal to the device, logic states are always active high. The output of an AND gate is only high if all connected inputs are high, and the output of an OR gate is high as long as at least one of the connected inputs is high. All symbolic logic inversion takes place only at the input and output buffers. This can get quite confusing with PAL devices, most of which do not have programmable output polarity. This makes it necessary to apply DeMorgan's Law to your logic equation if you want to change its output polarity (Figure 2). Fortunately, this is taken care of for you if you are using one of the several logic compilers available, which I'll briefly describe later.

$$\overline{(X + Y)} = \overline{X} \cdot \overline{Y}$$

$$\overline{(X \cdot Y)} = \overline{X} + \overline{Y}$$

Figure 2 - DeMorgan's Law

Figure 3 shows the logic representation of an input buffer. All signals are passed into the programmable array in both their normal and inverted states. This eliminates the need to have signals of the correct polarity available at the inputs to the device.

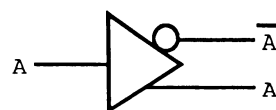


Figure 3 - Input buffer

Internal Architecture

PROMs, PALs, and FPLAs differ in the way that they implement the standard sum-of-products equation. These differences affect their flexibility, speed, and potential applications.

PROMs were the original programmable logic devices, and as such are the least flexible. They are also the least complex and the least expensive. MMI is attempting to resurrect PROMs as legitimate replacements for random logic by providing support for them in their logic compiler and by renaming them Programmable Logic Elements (PLE). This ploy was obviously dreamed up by the marketing department. A rose by any other name ...

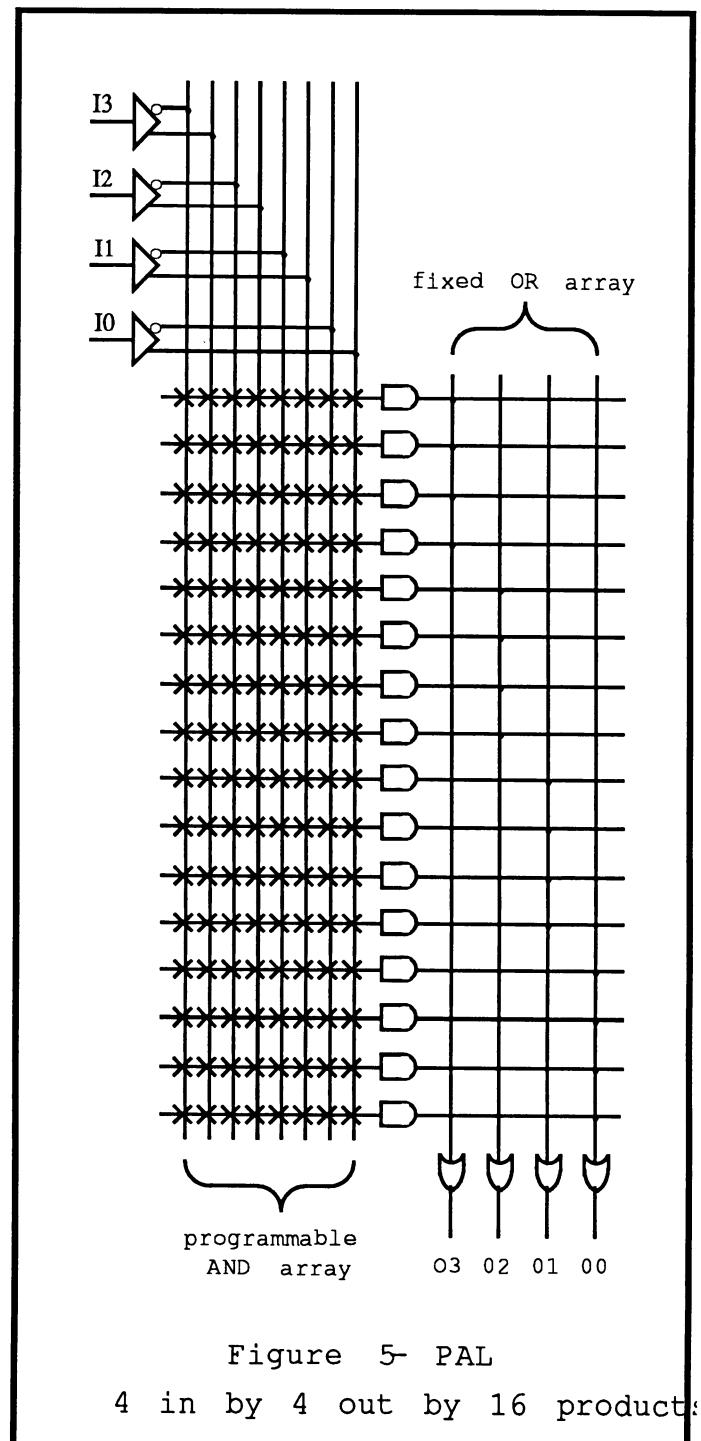
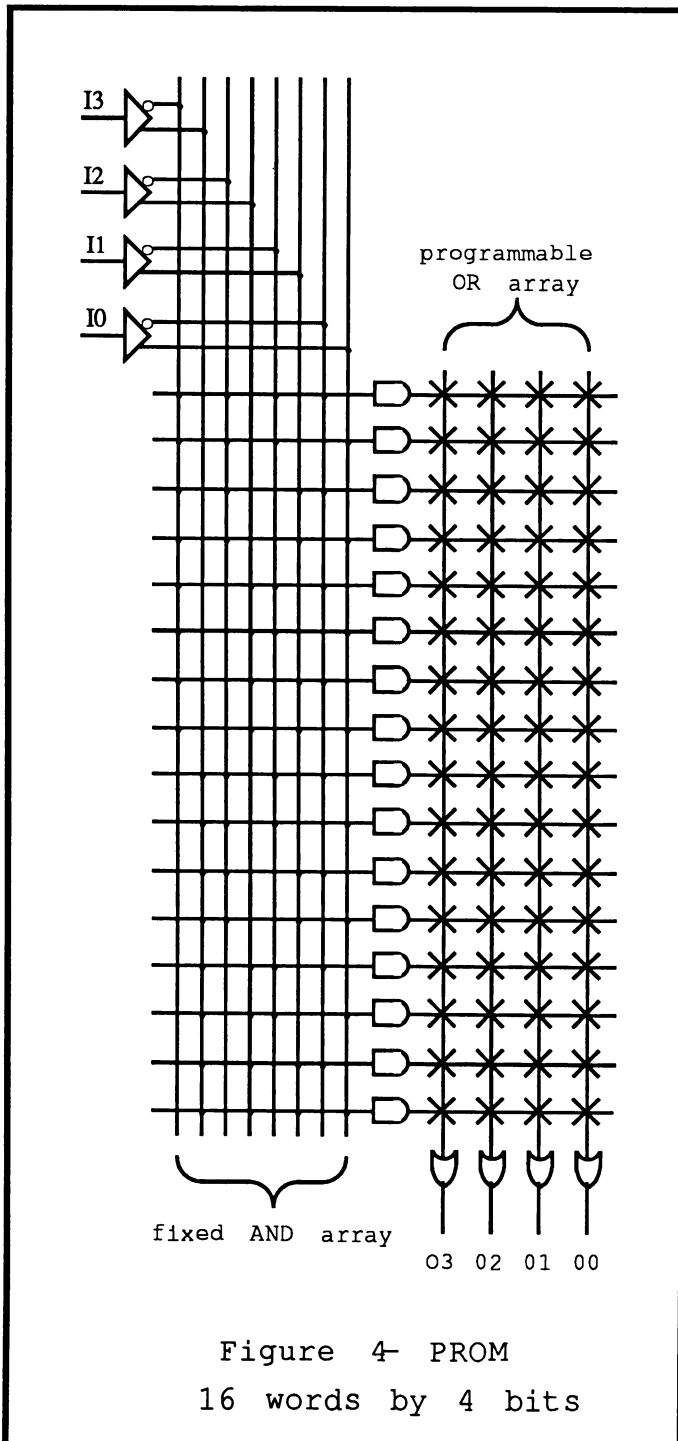
The internal structure of a PROM consists of a fixed AND array followed by a programmable OR array. This is shown in Figure 4. The most common use for a PROM is as a computer program store, where the inputs are addresses and the outputs are data. They are also used as lookup tables,

where the input would be a variable and the output is some function of that variable, for example output = SIN (input). PROMs are good for applications where every possible input combination has an associated output value, which is rarely the case with random logic.

The basic structure of a PAL, shown in figure 5, is very similar to that of a PROM. The difference is that now it is the OR array that is fixed and the AND array that is programmable. This architecture is much better suited to the

replacement of random logic.

The ultimate in flexibility is the FPLA. As you can see from figure 6, in this device both the AND array and the OR array are programmable. This provides the designer the capability to implement any logic function, as long as there are enough terms available. This extra capability extracts a price in increased device complexity, which makes the part more expensive. The additional fuses also increase the propagation delay through the device, since there is a capacitance associated with each unblown fuse. This



encourages designers who require speed to shy away from FPLAs, although if all the unused fuses are blown they are just as fast as PALs. Unfortunately, this is not reflected in the data sheets for FPLAs, so not everyone is aware of this.

The last type of programmable logic device is really not programmable at all. The HAL[®] (Hard Array Logic) is a custom PAL device. It is actually HALs that are inside the Macintosh, not PALs. If product volume warrants it, a semiconductor mask can be made from a PAL fuse map.

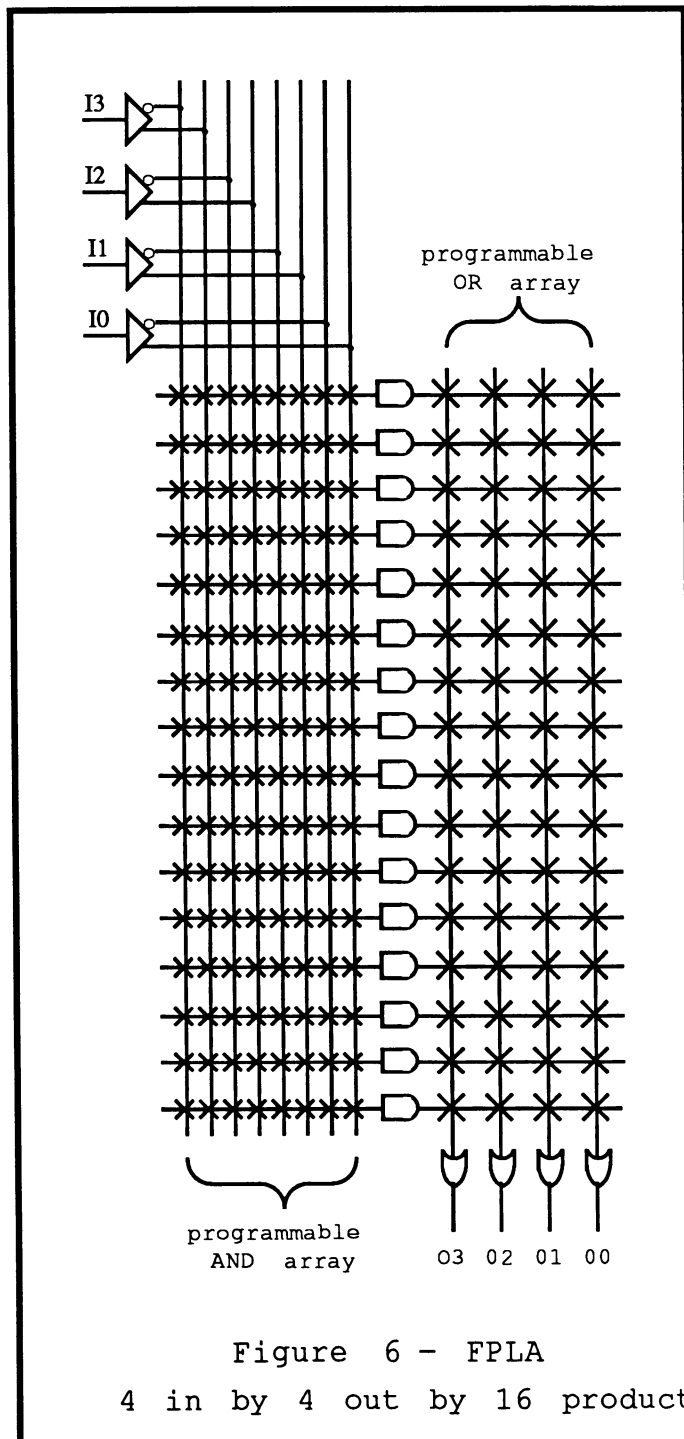


Figure 6 - FPLA
4 in by 4 out by 16 product

Custom devices are then made from these masks. A HAL is to a PAL as a ROM is to a PROM. This is quite a boon to the designer, who can prototype using PALs, then use HALs for high volume production. This is unlike gate arrays, which cannot be tested in an actual circuit until the design is done, masks are made, and production parts come off the line.

Programming

Once it has been decided what logic functions a PLD is to perform, how is the device programmed? There are several methods, but only a couple of practical ones. Most PROM programmers will, with additional attachments, program PLDs also. These programmers generally accept fuse-map data, which tells it which fuses are to be blown. Not being computers, we would prefer to input the data as boolean equations.

MMI was the first to fill the need for a 'silicon compiler' when they made PALASM[®] (PAL assembler) available. This is a FORTRAN IV program which allows the user to directly input boolean equations, from which it creates the fuse map which the programmer can understand. PALASM, being a product of MMI, only supports PALs.

Recently Data I/O, who manufactures programmers, and Structured Design, have each released second-generation compilers for programmable devices. These compilers (ABEL and CUPL, respectively) support all manufacturer's devices and provide more sophisticated facilities than were available on the original PALASM. They have macro capability, can do logic minimization, can simulate the target device, automatically generate test vectors, and are even device-independent, up to a point. Device independence means the same equations can be re-compiled for a different device. For example a PAL can be used in place of an FPLA, as long as the target device has the capability required by the application.

Application

I'll use the PAL structure in Figure 5 for an application example. Admittedly, the example is a bit simplistic, but it will demonstrate the basic sum-of-products principle.

The example circuit is shown in Figure 7. It is a 2 to 1 multiplexer with a strobe.

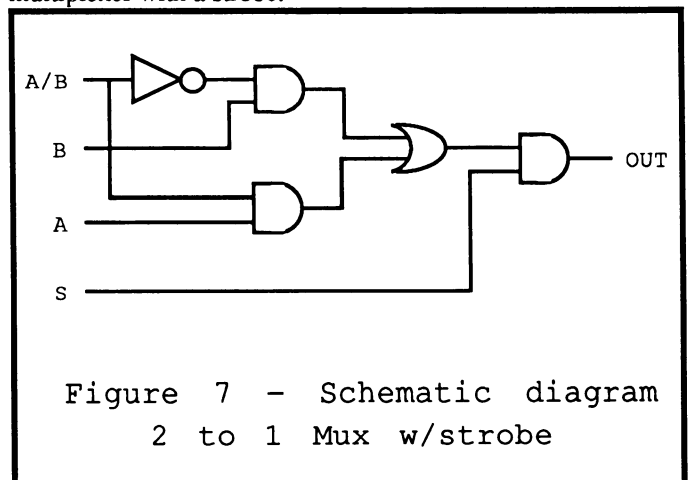


Figure 7 - Schematic diagram
2 to 1 Mux w/strobe

The truth table and logic equation is shown in Figure 8. The A/B input selects the A input when high, and the B input when low. The S input forces the output low unless it is high. The circuit in Figure 7 uses three types of gates, a 7404, a 7408, and a 7432. It uses 1/6 of the '04, 3/4 of the '08, and 1/4 of the '32, for a total of 1 1/6 14 pin packages. A PAL as a replacement for this logic would typically be a 20 pin package, but could replace much more than just this simple multiplexer. Figure 9 is the PAL in Figure 5 which has been programmed to perform the multiplexer function of Figure 7.

S	A/B	A	B	OUT
	X	X	X	0
1	0	X	0	0
1	0	X	1	1
1	1	0	X	0
1	1	1	X	1

$$OUT = (S \cdot A/B \cdot A) + (S \cdot \overline{A/B} \cdot B)$$

Figure 8 - Logic
2 to 1 Mux w/strobe

This example demonstrates the concept of programmable logic, and suggests that much more complicated logic substitutions are possible. In fact, PLDs are available with up to 64 inputs and 32 outputs, with registers (flip-flops) and feedback terms, and in a variety of technologies. CMOS and ultraviolet light erasable PLDs are just coming on the market in what was once a completely bipolar domain. The market for these devices is growing rapidly and they should soon be as familiar to everyone as EPROMS are now.

Programmable Logic in the Macintosh

The Macintosh uses six HAL devices, of four different types. These types are 16L8, 16R8, 16R6, and 16R4.

The 16L8 is very similar to the example I've given, with 8 dedicated input pins, and 8 pins which are programmable as inputs or outputs. The part number, 16L8, means there are 16 possible inputs, 8 possible outputs, and the outputs are active low. The active low outputs can be programmed as active high using DeMorgan's Law, as I described above.

The 16R8 is identical to the 16L8, only there is a D-type flip-flop (register) at the output of each OR term. Figure 10 shows the structure of one registered output term. The 16R6 and 16R4 have 6 and 4 output registers respectively, with the

remaining 2 and 4 terms direct combinatorial outputs like the 16L8.

Macintosh HAL Functions

A look at the functions that the Mac's HALs perform give a good feel for the type of applications where programmable logic is used. There are many instances of simple random-logic replacement, but there are also some clever applications that demonstrate the versatility of these devices.

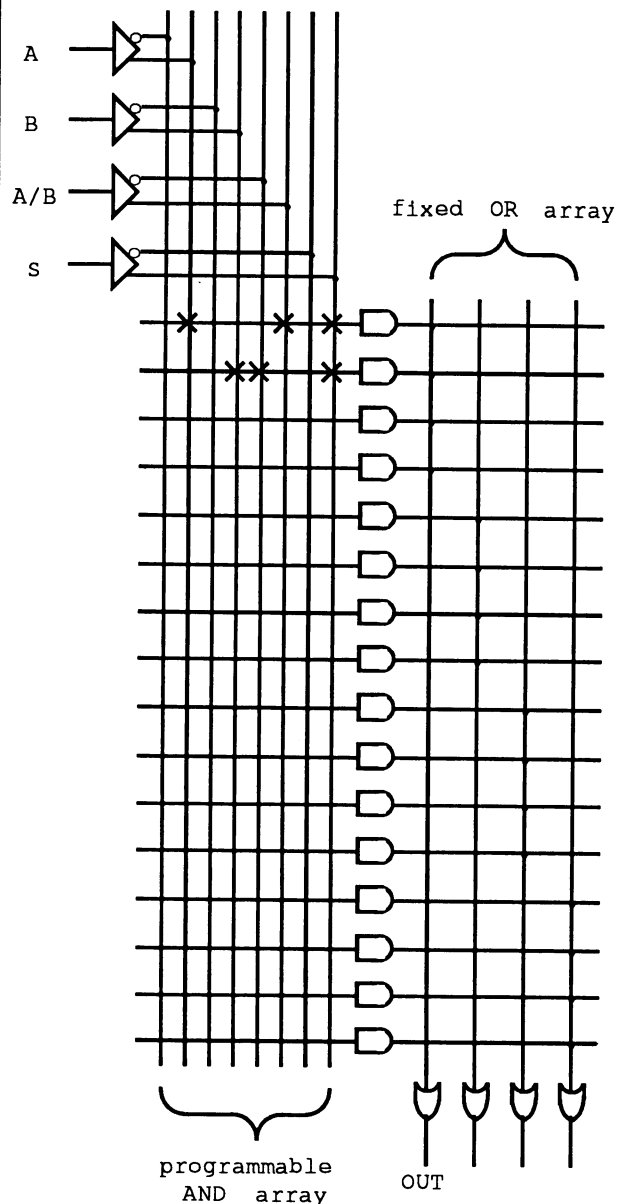


Figure 9- Example
2 to 1 Mux w/strobe

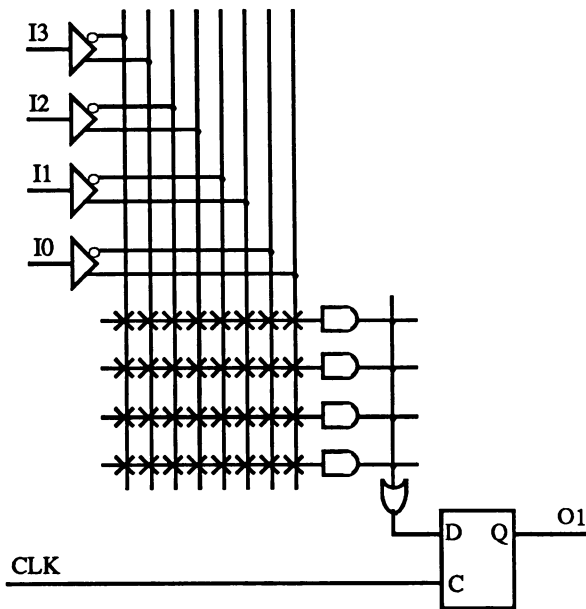


Figure 10- REGISTERED PAL

The 6 HALs in the Mac have designators silk-screened on the PC board next to them. I'll refer to them by those names, although I can't figure out what they stand for.

BMU1

BMU1 is a 16L8 device which performs the major address decoding functions. It has as inputs the higher order address lines A21, A22, and A23 from the processor, along with the overlay bit (for a description of the overlay bit, see the August issue). These bits are decoded to generate enable signals for the RAM, ROM, the IWM (disk controller) and the SCC serial chip.

LAG

LAG is a 16R8 device which performs the majority of the video control functions. It has as inputs most of the video address counter outputs, which are decoded to create output signals which load the video shift register, provide the CRT sweep circuitry with horizontal and vertical syncs, increment and reset the video address counters, and switch the RAM address multiplexers between CPU, video, and sound addresses.

BMU0

BMU0 is a 16R4 device which generates RAM read and write signals from the RAM enable output of BMU1 and the processor R/W line. It is also used as a counter to create two video address lines (VA12 and VA13) because the video

counter is only 12 bits wide (VA0-VA11). In addition, it also generates DTACK, the data transfer acknowledge handshake signal to the processor, and synchronizes the output of the video shift register with the master oscillator. This is a very good example of the many different types of functions that can be handled with a single programmable device.

TSM

TSM is a 16R4 device whose major function is control of the dynamic RAM. For inputs, it has the decoded RAM enable signal, along with the address and data strobes from the processor which signify whether the data transfer will be low byte, hi byte, or word. From these the RAS and CAS strobes are generated, and the row/column address multiplexer is controlled. Until there were PALs, this type of dynamic RAM control function required either about 10-20 discrete TTL packages, or a 40 pin LSI dynamic RAM controller which usually didn't do what you wanted anyway.

ASG

ASG is a 16R8 device which illustrates why PALs can be so valuable. Its primary purpose is to take the 6-bit disk speed value which is fetched at the end of every horizontal retrace period and convert it to a pulse-width modulated signal. Basically, it's a 6-bit counter. This leaves a couple of inputs and an output available, which are used to control the loading of the sound generator pulse-width modulator, which is a counter made up of TTL devices. If a discrete counter had been used for the disk PWM, another chip would have been required for the sound PWM load function. Using a PAL for a simple counter function in this instance saved a chip in the design.

TSG

This one is my favorite. The TSG is a 16R6 device which illustrates the power of programmable logic. It serves a couple of mundane functions concerning interrupts and the keyboard clock, but by far its most interesting job is as a clock generator for the SCC serial chip.

The master oscillator frequency in the Macintosh is 15.667 MHz. This is divided by 2 in the TSG to get the 7.834 MHz processor clock. In order for the SCC to be able to operate at a baud rate of 230.4 KBaud, which is what AppleTalk requires, it needs an input clock frequency of 3.686 MHz.

If you pull down your calculator desk accessory, you'll find that $15.667 \div 3.686 = 4.25$. This means that the TSG needs to divide the 15.667 MHz master oscillator by 4.25 in order to get a 3.686 MHz clock. How is this done, since 4.25 is not even an integer, let alone a binary number?

Let's call the 15.667 MHz clock the MO_clk and the 3.686 MHz clock the SCC_clk. For every 17 MO_clk periods there are 4 SCC_clk periods ($17 \div 4 = 4.25$). The way the TSG generates the SCC_clk is count to 4 three times and then count to 5 once ($4 + 4 + 4 + 5 = 17$). See Figure 11 for

a graphical description. Try that using a single TTL counter chip!

In reference to my opening paragraph, I hope by now everyone realizes that PALs aren't black magic. In fact, I'd term them white magic, since they are used not for evil, but for good.

PAL® and HAL® are registered trademarks and PALASM™ is a trademark of Monolithic Memories Inc.

REFERENCES: Programmable Array Logic Handbook, Monolithic Memories Inc., Santa Clara, CA 1981.

Integrated Fuse Logic Data Manual, Signetics Corp., Sunnyvale, CA 1984.

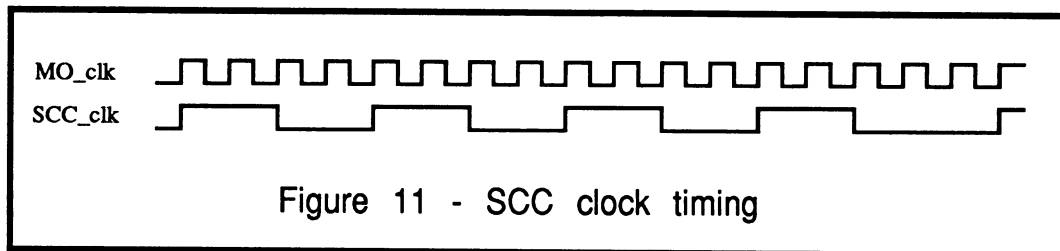


Figure 11 - SCC clock timing

Macintosh Serial Ports

In order for their concept of the 'virtual slot' to be accepted as a viable alternative to true hardware expandability, Apple had to forego the industry standard RS-232 in favor of an interface that would support a much higher data rate, thus the choice of RS-422. They also chose to compromise the standard somewhat in order to make the output voltages more compatible with RS-232.

For output drivers, the Macintosh uses 26LS30's running from ± 5 volt supplies, rather than the single +5 volt supply recommended by the manufacturer for RS-422 operation. The result is a symmetrical open-circuit voltage swing of about 8 volts centered around ground. This drops to about 4 volts when loaded with 100 Ω . Centering the output signal around ground makes it RS-232 compatible, but the 8 volt open circuit voltage exceeds the RS-422 restriction of a maximum of 6 volts.

The receivers are 26LS32's and meet all RS-422 requirements. Since the Macintosh handshake line is a single-ended input, the inverting input on that receiver is tied to ground, making that input RS-232 compatible. The TXD and RXD lines are differential but are compatible with RS-232 levels when connected to the inverting (-) terminals. The 26LS32 is designed so the non-inverting (+) input floats at about +2.7 volts when open and the inverting input floats at about +1.8 volts. Therefore, when an RS-232 signal is connected to the inverting input the receiver will switch around +2.7 volts if the non-inverting input is left open, which is within the ± 3 volt RS-232 spec.

Between the drivers/receivers and the connectors, the signal lines go through an RC filter network. This is not part of the RS-422 specification, but is included to meet FCC EMI requirements and to provide signal deglitching.

The Macintosh serial connector pinout is as follows:

PIN # FUNCTION

1	GND
2	+5 V
3	GND
4	TXD +
5	TXD -
6	+12 V
7	HSK
8	RXD +
9	RXD -

For RS-232 compatibility, use Gnd, TXD-, and RXD- (remember, RS-232 signals are inverted). The HSK line goes to both the CTS and the TRxC inputs on the 8530 chip in the Macintosh. Depending upon how the chip is configured by the serial driver, this line may be used as a handshake line or as an external clock in order to increase data transfer speeds. The +5 volt and +12 volt outputs come directly from the power supply inside the Macintosh. Apple warns that the +5 volt line may turn into an output handshake line at some future date and that it should not be used. They also warn that the +12 volt line is for detecting power only. When connecting peripherals or making cables, exercise caution, but you'll probably not be able to damage any of the interface circuitry. Pay special attention to the power pins (2 & 6), however, since these are tied directly to the power supply and are not protected from accidental shorting.

Electrically, like most of the Macintosh hardware, the serial ports are quite simple. Some compromises to the RS-422 standard were made in Apple's implementation, but generally speaking, these are not significant. The end result has been to provide users with a much more flexible interface, able to communicate with nearly all RS-232 peripherals in addition to supporting much higher data transfer rates for applications that require it, such as external disk drives and networking.



Ask Prof. Mac

Technical Questions Answered

Steve Brecher
MacTutor Contributing Editor
MacTutor Vol. 1 No. 11

[Prof. Mac is dedicated to helping you solve your Macintosh Programming problems. Each month Prof. Mac will research out your problem and print the best possible solution in MacTutor. Don't waste time in frustration! Send your technical questions to Prof. Mac and let him research your problem for you. Write to Prof. Mac, care of this Journal. -Ed.]

Coordinate Systems

Q. Moving a window by using `MoveWindow(..., <portRect.left+offset>, <portRect.top+offset>, ...)` doesn't work. What do I need to do?

A. The coordinates you pass to `MoveWindow` must be global coordinates, so transform them with calls to `LocalToGlobal` first. The distinction between local and global coordinates can be confusing (it was to me, anyway), so let's discuss coordinate systems.

A coordinate system is an abstraction. I visualize it as an infinitely large sheet of graph paper with one intersection of grid lines designated 0,0. By assigning one grid intersection (0,0 or any other) to a specific bit in memory, we thereby are able to refer to pixels (memory bits) in terms of coordinates rather than in terms of RAM addresses. This is very convenient when thinking about graphics.

The QuickDraw *bitmap* is what associates a particular coordinate system (graph paper) with a particular set of memory locations.

Think of the sheet of graph paper thumbtacked to a bit in memory. The thumbtack goes through the graph paper at the intersection designated by the top, left coordinates of the bitmap's bounds rectangle; this can be any intersection at all - it needn't be 0,0. The tack's point sticks into memory at the location designated by the bitmap's `baseAddr`.

A point expressed in local coordinates is merely one that has been thus assigned to a pixel as specified by the relevant `grafPort`'s *bitmap*. Each and every coordinate found in a `grafPort` (in the sense of a field in a Pascal record) is, by definition, local to that `grafPort`.

When we wish to compare the locations with respect to memory (screen) of points in two different `grafPorts`, we must adjust their coordinate systems so that the same intersection on each port's graph paper is assigned ("thumbtacked") to the same pixel. The convention for doing this is to "move" each graph paper so that its 0,0 coordinate is at the lowest-addressed

pixel. Note that this method of inter-port comparison works only if both lowest-addressed pixels (`baseAddr`'s) are the same! This implied requirement -- satisfied, of course, if both `baseAddr`'s are equal to `screenBase` -- is usually taken for granted in IM discussions.

A local coordinate is an expression of vertical and horizontal distance from the 0,0 intersection on the particular piece of graph paper associated with a `grafPort`. A global coordinate is an expression of vertical and horizontal distance from the `baseAddr` pixel. If (and only if) two ports have the same `baseAddr`, then global coordinates from each may be compared and yield a valid graphics relationship.

While the QuickDraw chapter of *Inside Macintosh* says (of two ports being compared) "using the same bit image (such as the screen)", the rest of IM when using the term "global" takes for granted that the screen is in fact the common bit image for both ports.

When IM says a document being drawn "sticks to the coordinate system," I mentally translate that to "sticks to the graph paper"; similarly, I mentally translate "sticks to the screen" to "sticks to memory."

Of course, the memory locations designated by a *bitmap* do not have to coincide with the memory from which the screen is displayed. Drawing merely affects the memory designated by the *bitmap*; if that memory is (all or partially) in the screen buffer, then the screen display will be affected.

QuickDraw Regions Limitation

Q. L. Tannenbaum of Long Beach, CA, submitted some MacFORTH code that illustrates a problem with QuickDraw's handling of complex regions. His program created 17 vertically-oriented rectangles while defining a region. Subsequently `FrameRgn` didn't seem to work correctly. He wants to know if the problem is in his code or in QuickDraw.

A. By doing some experiments, and with the help of MacsBug, I analyzed the problem as follows.

There is a problem in QuickDraw's handling of regions containing more than 12 vertical (higher than wide) rectangles. `FrameRgn` of a region containing more than 12 vertical rectangles will paint them instead of frame them, and rectangles after the 12th (from left to right) will be enlarged horizontally.

If there are more than 24 such Rectangles, `FrameRgn` will crash with an address error.

The problem seems to be due to the way region information is stored, in conjunction with the fact that some routines (e.g., `FrameRgn`, possibly as a part of code

common to other routines) allocate a fixed amount of space on the stack which is not large enough to accommodate a chunk of information recorded for the region.

For (at least) multiple rectangles, region information is stored in chunks consisting of pairs of arrays of coordinates in the form

```
top[i]    left[1] right[1] left[2] right[2] ... left[n] right[n]
bottom[i] left[1] right[1] left[2] right[2] ... left[n] right[n]
```

where *i* ranges over 1 to the number of horizontal lines on which the rectangle corners lie, from top to bottom of the `grafPort`'s `portrect`. Each array is terminated with a `$7FFF` marker.

For horizontally-oriented rectangles, *n*=1, and each array contains exactly 3 coordinates. For vertically-oriented rectangles, *n* is equal to the number of rectangles, and each array is therefore potentially large. QuickDraw appears to do a Link A6, #-562, and part of the stack frame thus allocated appears to be filled with a chunk of coordinates via autoincremented A2 as the destination address of a move loop. If the chunk is too large, stack frame underflow will result.

I submitted the above to Apple's Tech Support team, and Ginger Jernigan replied:

"It isn't a bug, it is a limitation in QuickDraw, which will be documented in the final version of Inside Macintosh. Your analysis of the situation was correct. We're working to fix it or at least alleviate the nasty problems that occur (like getting real syserrors from Quickdraw calls)."

Boldly Outlining Buttons

Q. Mike Scanlin asks: how do you get the thick border to outline a button in a dialog box for the button representing the default if the user hits Return (usually either "OK" or "Cancel")?

A. Usually such button highlighting appears in alert boxes. The Dialog Manager automatically highlights either item 1 or item 2 in an alert box. Whether item 1 or item 2 is highlighted depends on the value of a bit in the "stages" word for the current stage of the alert. (See *Inside Macintosh*, Dialog Manager, pp. 34-36.)

The stages word is located in the `ALRT` resource, and is divided into four 4-bit parts, each part governing one stage, or consecutive occurrence, of the alert. One bit in each part governs item highlighting; if the bit is clear, item 1 is highlighted, and if it's set, item 2 is highlighted.

Often item 1 is an "OK" button and item 2 is a "Cancel" button, and *IM* assumes this in its illustration. But in fact the Dialog Manager will highlight item 1 or item 2 regardless of what kind of item it is. This can sometimes be a problem. For example, if you have an alert with only a message (a `statText` item) and an "OK" button, you don't want *any* highlighting. But the Dialog Manager will relentlessly highlight either item 1 or item 2; if the `statText` is item 2 and the appropriate bit in the stages word is set, the text will have

an ugly bold round-corner rectangle drawn around it. If the bit is clear, the "OK" button will be highlighted, which would be silly since it's the only enabled item.

The only way to handle such a situation is to add a dummy item to the alert's item list (such as a one-character `statText` item located outside of the alert's rectangle) and let that dummy item "take" the highlighting. In our example, the dummy item could be item 1, and the bits in the stages word would be clear so that item 1 is highlighted. To see an example of this technique, use ResEdit to examine the Finder's DITL 129 resource.

To highlight a button in a dialog that is not an alert, you can use a `userItem`. Usually a `userItem` just draws a non-standard item, such as a picture. In this case, we can employ a `userItem` to change the appearance of *another* item -- namely, of our button.

A `userItem` consists of a procedure, as documented in *IM*, Dialog Manager, p. 11. We will ignore the `itemNo` parameter, since the item that the `userItem` will be operating on is not the `userItem` itself, but the button we want to outline. The QuickDraw techniques for boldly outlining a button are shown on p. 13 of the Dialog Manager section:

```
PenSize(3,3);
InsetRect(displayRect,-4,-4);
FrameRoundRect(displayRect,16,16)
```

--where `displayRect` is the button's display rectangle; `GetDItem` can be used to obtain it.

Print Dialogs

Q. Paul Cozza of Long Beach, CA, asks: two undocumented Printing Manager routines -- `PRSTLINIT` and `PRJOBINIT` -- were mentioned on p. 32 of the First Draft (6/11/84) of Printing Manager section of *Inside Macintosh*. Apple promised documentation in a later release of the manual, but the routines have mysteriously disappeared altogether from the latest version (Second Draft, 3/27/85, distributed with the May 1985 Software Supplement). If these routines are now unavailable, what is the way to modify the print style and job dialogs?

A. My guess is that the routines succumbed to the generalization of printing procedures that was necessitated by the advent of the Laserwriter.

In the Imagewriter file (I'll let you know about the Laserwriter as soon as I can afford one!), the style dialog is the `DLOG -8192` resource, and the job dialog is `DLOG -8191`. If you examine `DITL` (dialog item list) -8192, you'll see only placeholders where the paper sizes appear in the style dialog.

At least with respect to Imagewriter paper sizes, it is possible to make some changes. The `PREC 3` resource in the Imagewriter file specifies the names of the papers as they will appear in the style dialog, and the sizes of the paper. The format of the `PREC 3` resource is as follows:

n [1-word count] -- number of paper sizes;
n pairs of words -- vertical, horizontal paper size in
 120ths of an inch;
n Pascal-format strings -- names of paper sizes;
1 word -- ? (probably flags for Orientation, Pagination,
 Reduction).

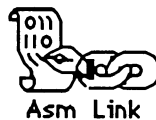
Up to six paper sizes can be accommodated by the style dialog box. The distributed Imagewriter file PREC 3 resource contains the specifications of five sizes (US Letter, A4 letter, US Legal, International Fanfold, Computer Paper).

As to other aspects of the dialogs I'm afraid you're on your own; I note the following warning in *IM*:: "Your application should not change the data in the print record -- be sure to use only the standard dialogs for setting this information."



Assembly Language Lab

The Talking Mac



Dan Weston
MacTutor Vol. 1 No. 12

In the May, 1985, issue of the Macintosh Software Supplement, Apple released a package of tools and code units collectively called MacinTalk 1.1. With these tools, programmers can make their Macintosh programs talk without any additional hardware. In this article I'll explain the general workings of MacinTalk and develop a small application program in assembly language that will show you how to use the main features of MacinTalk in your own programs.

Overview of MacinTalk

The MacinTalk system's most basic component is a driver that contains several procedures available to your programs. The driver is contained in a file called 'MacinTalk', and this file must be on the same volume as any application that wishes to use the MacinTalk driver. The most basic function of the driver is to convert ASCII strings of phonetic codes into speech. You can also use another part of the driver to convert standard English text into phonetic codes which can then be spoken by the driver. Furthermore, there are parts of the driver that you can use to control the rate of speaking and the pitch.

Beyond the actual driver procedures that you will be using in your programs, there are a few tools that are useful to you while you are preparing a program that will use speech. The program 'Speech Lab' allows you to enter English text in one window and then hear the MacinTalk speech and see the phonetic translation in another window. This program is very useful for learning the tricks of the phonetic code system used by MacinTalk. For example, the English sentence "This is a test." is translated into the phonetic string, "DHIHS IHZ AH TEHST.#". This program can be used to pre-translate strings that your program will speak when the strings are known ahead of time. It is more efficient, both in time and memory, to feed phonetic strings directly to the MacinTalk driver rather than relying on translation at run time. Also, if you pre-translate you will be able to fine tune the phonetics, because the translation is not always perfect.

The translation of English to phonetics is governed by hundreds of phonetic and grammatical rules contained in the MacinTalk driver, but these rules will not get every word right. Another program in the MacinTalk 1.1 package is 'Exception Edit'. This program allows you to create a special file of tricky words and their correct phonetic translation. Exception Edit lets you experiment with the phonetic strings until you get them right, and then save those translations for later use. A file created by Exception Edit can be automatically loaded and utilized by mentioning it when the MacinTalk driver is

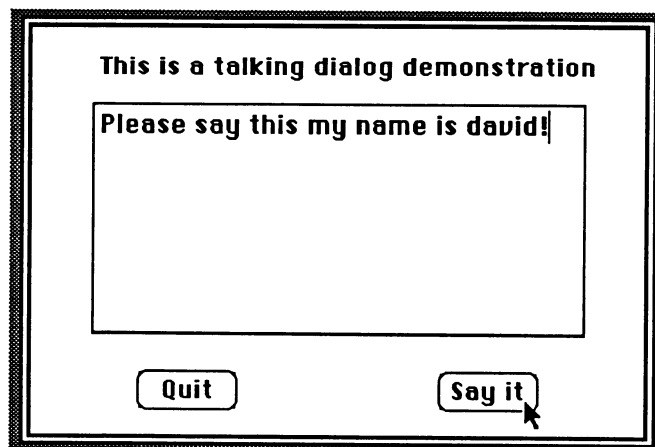


Fig. 1 Program Output

opened, as shown in a later section of this article.

The MacinTalk Driver

There are seven procedures in the MacinTalk driver that your program can call. They are listed briefly below.

**FUNCTION SpeechOn(ExceptionsFile: Str255;
theSpeech: SpeechHandle): SpeechErr;**

This function opens up the driver and initializes the values for speed and pitch. If you pass a null string for ExceptionsFile, then the translation of English to phonetics will follow the standard rules. If you pass a valid file name for ExceptionsFile, then that file, which must have been created by Exception Edit, will be used to help guide translation. If you pass the string 'noReader' for ExceptionsFile, then the driver will be opened but it will only be able to receive phonetic input and it will not be able to translate English to phonetics.

**PROCEDURE SpeechOff(theSpeech:
SpeechHandle)**

This procedure closes the driver and deallocates any storage that it has been using.

**FUNCTION MacinTalk(theSpeech: SpeechHandle;
Phonemes:Handle): SpeechErr**

This is the work horse of the driver. This is where

phoneme code strings are converted to speech. The handle to the phonemes should refer to a string of ASCII phonemes without a length byte.

FUNCTION Reader(theSpeech: SpeechHandle;
EnglishInput: Ptr; InputLength: LongInt;
PhoneticOutput: Handle): SpeechErr

This is where English strings are translated into phonetic strings that can then be fed to MacinTalk. The Ptr to EnglishInput should not point to a length byte of a Str255. Point to the first character instead. The Handle for PhoneticOutput can start out as a zero length Handle, and Reader will dynamically grow the Handle to fit the output.

PROCEDURE SpeechRate(theSpeech:
SpeechHandle; theRate:INTEGER)

This sets the rate at which words are spoken, in words/min. The rate must be between 85 and 425 words/min.

PROCEDURE SpeechPitch(theSpeech:
SpeechHandle; thePitch: INTEGER; theMode:
FOMode)

This sets the baseline pitch, in Hz, and sets the pitch mode, either natural or robotic.

PROCEDURE SpeechSex(theSpeech:
SpeechHandle; theSex:Sex)

This is not implemented in MacinTalk 1.1

The glue which calls the various procedures in the driver is contained in the file SpeechASM.Rel, also available in the Software Supplement. Make sure that you include SpeechASM.Rel in the link file for your application so that the driver routines will be available to your code. Also, you must XREF the individual routines that you wish to use. See the listings of CheapTalk.ASM and CheapTalk.LINK for examples.

CheapTalk: A Simple Speech Application Example

The software supplement contains the source code for a very short example program that shows how to use the speech driver. As usual, it is in Pascal, so we assembly language programmers have to muddle along and figure things out ourselves. In order to learn the system myself, and to provide a clear example of the main features of MacinTalk, I have written CheapTalk, a dialog-based application that speaks pre-translated text stored in a resource file and also translates and speaks user input at run time. CheapTalk opens a dialog and speaks the static message one time. Then it waits for the user to type English text into an edit text box in the dialog. Hitting return or pressing a 'Say it' button will translate the English text into phonemes and then say it.

This application will show you how to open and close the driver, and how to use MacinTalk and Reader from assembly language. It does not use the procedures to control the speed of pitch, but I imagine that you can figure that out for yourselves.

In my discussion of the code, listed in listing 1 as CheapTalk.ASM, I will concentrate on the parts pertinent to MacinTalk, and leave many of the details of the shell to speak for themselves.

Making the Connection to SpeechASM.Rel

Toward the beginning of CheapTalk.ASM, notice the XREF statements necessary for the linker to establish the connection between our routine calls and the SpeechASM.Rel code that we link with our code.

XREF	SpeechOn	; open driver
XREF	MacinTalk	; speak phonetic string
XREF	Reader	; translate English to phonetics
XREF	SpeechOff	; close driver

The linker control file is listed in listing 2 as CheapTalk.LINK. SpeechASM.Rel is a code file which contains the glue routines necessary to call the individual procedures contained in the driver. SpeechASM.Rel does not contain the actual speech routines, just short procedures to call the appropriate section of the MacinTalk driver. All the routines of the speech driver expect their parameter on the stack.

Setting Up the Global Variables for Speech

Next, notice the global variable, 'theSpeech', defined as a long word to hold the handle to the speech globals that will be allocated when the driver is opened. We only have to define a variable to hold the handle, the opening routine will allocate the necessary storage for the speech globals. Other globals that we need to define include a word length flag that we use to show if the driver was successfully opened, a 256 byte block to hold an English string, and a handle which will be used for phonetic output from Reader.

theSpeech	DS.L	1	; handle to speech driver globals
speechOK	DS.W	1	; our flag to show if driver opened
theString	DS.B	256	; keep our English string here
phHandle	DS.L	1	; handle to phonetic string

If you look at CheapTalk.ASM you will see that there are several other global variables defined to use as VAR parameters associated with maintaining the dialog box.

Opening the Driver

When we call SpeechOn to open the driver, we specify the null string (a string with length 0, which we define in the static variable area at the end of the code) for the

ExceptionsFile so that the Reader will translate English to phonetics using the default rules. If we had a specific exceptions file that we had created with Exception Edit, then we could pass in that file name so that the exception file would be used. We also pass the address of our global variable, theSpeech, so that it can be updated to hold the handle to the speech globals which will be allocated by the open routine.

```
; assume that driver will open alright, set our flag to TRUE
MOVE.W #1,speechOK(A5) ; set flag to TRUE
; now open driver to use default rules for translation
;FUNCTION SpeechOn(ExceptionsFile:Str255;
; theSpeech:SpeechHandle): SpeechErr
;
CLR.W -(SP) ; space for result
PEA NULL ; defined at end of code
PEA theSpeech(A5) ; handle for speech global
JSR SpeechOn ; jump to open routine
MOVE.W (SP)+,D0 ; get result code
BEQ @1 ; branch if open OK
```

```
; if driver open not successful then clear speechOK flag
; to prevent further use of invalid driver
MOVE.W #0,speechOK(A5) ; set flag to FALSE
; you could also display an error dialog here
```

```
@1 ; branch to this point if open is successful
```

You can see how the result code is checked after SpeechOn to see if the driver was opened successfully. In the event of a non-zero result, implying a problem with the opening, we set the speechOK flag to 0 and continue on with the program. All other parts of the program which use the speech driver first check the speechOK flag to make sure that there is a valid driver to work with.

Speaking Pre-translated Speech

The static message in our dialog box is "This is a talking dialog demonstration." There is a phonetic translation of that string kept in the resource file as a resource of type PHNM. The translation was done using Speech Lab, and the resulting phonetic string put into the RMaker source file, listed in listing 3 as CheapTalk.R. I created the PHNM resource type for RMaker so that the phonetic string would not have a length byte. As a general strategy you can translate the static message of any dialog into a PHNM resource with the same resource ID number as the dialog. That way, it is easy to display the dialog and speak the message together.

When the PHNM resource is loaded into memory by GetResource, you get a handle to the phoneme string that you can pass to MacinTalk to recite. Remember, no length byte on phonetic strings! Generally, you should to pre-translate any strings that you know at assembly time, so as not to waste time and memory translating at run time, and also to insure higher quality speech by testing and refining the phonetic strings. Look at the following code to see how the

PHNM resource is retrieved and then fed to MacinTalk.

```
; first check our flag to make sure that driver is open
TST.W speechOK(A5)
BEQ @2 ; driver not valid
; branch around speech stuff
; driver valid, go ahead and speak
;FUNCTION
; GetResource(theType:ResType;ID: INTEGER): Handle
CLR.L -(SP) ; space for result
MOVE.L #'PHNM',-(SP) ; resource type PHNM
MOVE.W #theDialog,-(SP) ; use same ID# as dialog
_GetResource
MOVE.L (SP)+,A0 ; handle to phoneme string

;FUNCTION MacinTalk(theSpeech:SpeechHandle;
; Phonemes:Handle):SpeechErr
CLR.W -(SP) ; space for result
MOVE.L theSpeech(A5),-(SP) ; speech global handle
MOVE.L A0,-(SP) ; handle to phonemes
JSR MacinTalk ; say it
MOVE.W (SP)+,D0 ; get result code
```

```
@2 ; branch to here to avoid speaking with invalid driver
```

Translating English to Phonetics and then Speaking

After saying the static dialog message upon opening, the program waits for the user to enter English text in the edit text window of the dialog. The program watches the results of ModalDialog until the 'Say it' button is pushed, at which point it uses GetDItem and GetDText to get the current English text of the edit text item. That text, which is a Str255, is fed into Reader to translate it into a phonetic string. Please notice that when we pass the English text into Reader, we skip over the length byte at the head of the Str255. We do, however, use the length byte, after coercing it to a long word, as the length input to Reader. The Handle which we use to hold the phonetic output of Reader is initially associated with a zero length block, but Reader grows the block automatically to fit the output. Look at this code fragment which feeds the English string to Reader. (Assume that the string has already been placed in the variable 'theString' by calls to GetDItem and GetDText.)

```
; set up an empty handle first for Reader to fill with phonemes
;FUNCTION NewHandle(logicalSize: Size): Handle
; logicalSize => D0, Handle => A0
MOVEQ #0,D0 ; set up empty handle
_NewHandle
MOVE.L A0,phHandle(A5) ; save Handle for later

;FUNCTION Reader(theSpeech:SpeechHandle;
;EnglishInput:Ptr;
;InputLength:LongInt: PhoneticOutput:Handle):: SpeechErr
CLR.W -(SP) ; space for result
MOVE.L theSpeech(A5),-(SP) ; speech globals
```



```

PEA    theString+1(A5)    ; Ptr to string, skip length
CLR.L   D0                ; clear out D0
MOVE.B  theString(A5),D0  ; put length byte in D0
MOVE.L  D0,-(SP)          ; use longInt for length
MOVE.L  phHandle(A5),-(SP); we just allocated this
                        ;handle
JSR     Reader            ; do translation
MOVE.W  (SP)+,D0          ; get result

```

Once we have used Reader to translate the English text into a phonetic string, we pass the handle to the phonemes to MacinTalk, much as we did earlier, to hear it spoken. Here is the code which speaks the translation and then deallocates the handle which held the phonetic string. It is important to deallocate this handle after the phonemes are spoken to avoid cluttering up memory with old sayings.

```

;FUNCTION   MacinTalk(theSpeech: SpeechHandle
;Phonemes: Handle):SpeechErr
  CLR.W    -(SP)          ; space for result
  MOVE.L   theSpeech(A5),-(SP) ; speech globals
  MOVE.L   phHandle(A5),-(SP); handle to phonemes
  JSR      MacinTalk       ; say it
  MOVE.W   (SP)+,D0        ; get result
; deallocate handle
;PROCEDURE DisposHandle(h: Handle)
; h => A0
  MOVE.L   phHandle(A5),A0 ; where phonemes are
  _DisposHandle

```

This process can be generalized to other situations where you want to translate arbitrary English text into speech. Just get a pointer to the first character of the text, get the length of the text, allocate an empty handle, and feed it all to Reader. The phonetic output of Reader can then be handed to MacinTalk to recite.

Closing the Driver

We merely make a call to SpeechOff with theSpeech as input to close up the driver and deallocate the memory used by it. Generally, Macintalk will use at least 20K of memory, plus dynamic buffers equal to about 800 bytes/second of uninterrupted speech (usually less than 10 seconds). In addition, Reader utilizes 10K plus a buffer to hold the translated text.

```

;PROCEDURE SpeechOff(theSpeech: SpeechHandle)
  MOVE.L   theSpeech(A5),-(SP) ; handle to speech
                        ;globals
  JSR      SpeechOff          ; close it up

```

Putting It All Together

Listings 1, 2, and 3 show the assembler source file, the linker control file, and the RMaker source file. You should assemble CheapTalk.ASM, then link it with

CheapTalk.LINK. One thing to notice about the output file from the linker is that it is not a functional application until it is combined with the necessary resources by RMaker. Since Link output files are normally application type file, CheapTalk.LINK assigns a file type of 'CODE' so that the resulting output file will not have the characteristic diamond shaped icon. The final step of the program development is to run CheapTalk.R through RMaker to create the DLOG, DITL, and PHNM resources and combine them in one application file with the output file from the linker. The output of RMaker, Cheap Talk, will be a independent application program which can be moved to any disk and run as long as the driver file, MacinTalk, is also on that disk.

Summary

This discussion has been rather superficial. You are encouraged to study the source code and steal whatever parts of it you find useful for your own applications. All parts of the MacinTalk system are available in the Software Supplement or in the DL8 area of the Mac Developers interest group (PCS-7) on Compuserve, including the MacinTalk 1.1 documentation that Apple provides. This documentation is a good place to learn more about the phonetic symbols that MacinTalk uses and some of the finer points of the available routines. You should also be aware that there is a licensing fee if you distribute programs that use MacinTalk 1.1, so contact Apple before you start shipping disks with MacinTalk on them.

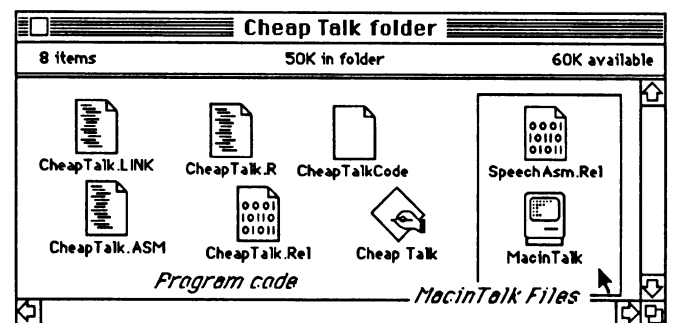


Fig. 2 Program files

```

; CheapTalk.ASM
; A short program to demonstrate how to
; use Macintalk 1.1 from assembly language

; This program displays a dialog and speaks
; the written message in the dialog

; It also will speak English strings written
; into an edit text box in the dialog

; copyright August 1985
; Dan Weston

```

```
; This program uses subroutines from the file SpeechASM.rel
; You must include that file in your link file list
; and XREF the particular routines here
```

```
; You must also have the file 'MacinTalk' on the same volume
; as this application program
```

```
XREF    SpeechOn           ; open driver
XREF    MacinTalk          ; say something
XREF    Reader             ; translate English to phonemes
XREF    SpeechOff          ; close the driver
```

```
theDialog    EQU    1      ; resource ID # of dialog
sayitbutton  EQU    1      ; item # for 'say it'
quitbutton   EQU    2      ; item # for 'quit'
usertext     EQU    3      ; item # for edit text box
```

```
INCLUDE      Mactraps.D
```

```
; ----- Global Variables -----
```

```
theSpeech    DS.L    1      ; handle to speech driver globals
speechOK     DS.W    1      ; our flag to show if driver open
theString    DS.B    256    ; VAR for GetlText
phHandle     DS.L    1      ; handle to phonetic string
```

```
ItemHit      DS.W    1      ; VAR for modal dialog
theType      DS.W    1      ; VAR for GetDItem
theltem      DS.L    1      ; VAR for GetDItem
theRect      DS.W    4      ; VAR for GetDItem
```

```
; ----- Initialization -----
```

```
BSR    InitManagers      ; at end of source file
```

```
; ----- Open the Speech Driver -----
```

```
; Open speech driver to use default rules
```

```
; assume that driver will open alright, set our flag to TRUE
```

```
MOVE.W #1,speechOK(A5)      ; set flag to TRUE
```

```
CLR.W -(SP)                ; result
PEA    NULL                 ; defined at end of source code
PEA    theSpeech(A5)        ; VAR theSpeech
JSR    SpeechOn             ; jump to to open routine
MOVE.W (SP)+,D0             ; check result
BEQ    @1                   ; branch if ok
```

```
; If driver open not successful then clear speechOK flag
; to prevent further use of invalid driver
```

```
MOVE.W #0,speechOK(A5)
```

```
; You could also put an error dialog here
```

```
@1      ; branch to this point if open is successful
```

```
;----- Get the Dialog from the Resource file --
```

```
CLR.L -(SP)                ;Clear Space For DialogPtr
MOVE   #theDialog,-(SP)    ; Resource #
CLR.L -(SP)                ;Storage Area on heap
MOVE.L #-1,-(SP)           ;Above All Others
_GetNewDialog              ;Get New Dialog
MOVE.L (SP)+,D6            ;Move Handle To D6
```

```
;PROCEDURESetPort (gp: GrafPort)
MOVE.L D6,-(SP)           ;Move Dialog Pointer To Stack
_SetPort                  ;Make It The Current Port
```

```
; usually you would not use DrawDialog, but we need to draw
; the dialog contents once before saying them, then go to
; Modal dialog which will draw the contents again
```

```
;PROCEDURE      DrawDialog(dp:DialogPtr)
MOVE.L D6,-(SP)
_DrawDialog
```

```
;----- Speak pre-translated speech -----
```

```
; now Say the static text item which has been pre-translated
; into a phoneme string with the same ID as the dialog
```

```
; first, check our flag to make sure that driver is open
```

```
TST.W speechOK(A5)
BEQ    @2                   ; driver not valid
                                ; branch around speech stuff
; driver valid, go ahead and speak
```

```
CLR.L -(SP)                ; space for result
MOVE.L #'PHNM',-(SP)       ; resource type PHNM
MOVE.W #theDialog,-(SP)    ; use same ID as dialog
```

```
_GetResource
MOVE.L (SP)+,A0            ; handle to phoneme string
```

```
CLR.W -(SP)                ; space for result code
MOVE.L theSpeech(A5),-(SP) ; speech global handle
MOVE.L A0,-(SP)            ; phonemes, from above
JSR    MacinTalk            ; say it
MOVE.W (SP)+,D0            ; get result code
```

```
@2      ; branch to here to avoid speaking with invalid driver
```

```
;----- Dialog loop -----
```

```
; now process the dialog
```

```
dialogloop
```

```
;PROCEDUREModalDialog (filterProc: ProcPtr;
;                      VAR itemHit: INTEGER)
CLR.L -(SP)                ;default filter proc
PEA    ItemHit(A5)          ;Item Hit Data
```

_ModalDialog

```
; see which button was pushed
CMP.W #quitbutton,itemHit(A5) ; quit button?
BEQ closeit

CMP.W #sayitbutton,itemHit(A5) ; say it?
BEQ sayit

; none of the above
BRA dialogloop
; go around again

;----- Translate English to Phonetics and speak -----
sayit

; first, check our flag to make sure that driver is open

TST.W speechOK(A5)
BEQ @3 ; driver not valid

; branch around speech stuff
; driver valid, go ahead and speak
; get the current text in the edit text box

MOVE.L D6,-(SP) ; we saved DialogPtr here
MOVE.W #usertext,-(SP) ; the edit text item
PEA theType(A5) ; VAR type
PEA theItem(A5) ; VAR item
PEA theRect(A5) ; VAR box
_GetDItem

;PROCEDUREGetIText(item:Handle;VAR text: Str255)
MOVE.L theItem(A5),-(SP) ; result of GetDItem
PEA theString(A5) ; VAR text
_GetIText

; now feed the text into reader to translate it into phonemes
; set up an empty handle first for Reader to fill with phonemes

;FUNCTION NewHandle(logicalSize: Size): Handle
; logicalSize => D0, Handle => A0
MOVEQ #0,D0 ; set up empty handle
_NewHandle
MOVE.L A0,phHandle(A5); save Handle for later

CLR.W -(SP) ; space for result
MOVE.L theSpeech(A5),-(SP) ; speech globals
PEA theString+1(A5) ;Ptr to string, skip length byte
CLR.L D0 ; clear out D0
MOVE.B theString(A5),D0 ; put length byte in D0
MOVE.L D0,-(SP) ; use LongInt for length
MOVE.L phHandle(A5),-(SP) ; we just allocated this
JSR Reader ; do translation
MOVE.W (SP)+,D0 ; get result

; now feed the phonemes to Macintalk

;FUNCTION ;MacInTalk(theSpeech:SpeechHandle;Phonem
es:Handle)
```

```
;
CLR.W -(SP) ;SpeechErr
; space for result code
MOVE.L theSpeech(A5),-(SP) ; speech globals handle
MOVE.L phHandle(A5),-(SP) ; handle to phonemes
JSR MacInTalk ; say it
MOVE.W (SP)+,D0 ; get result code

; deallocate handle and loop back for more

; PROCEDURE DisposHandle (h: Handle)
; h => A0
MOVE.L phHandle(A5),A0; this is where the phonemes are
_DisposHandle

@3 ; branch to here to avoid speaking with invalid driver

BRA dialogloop

;----- Close up shop -----

closeit
;PROCEDURECloseDialog (theDialog: DialogPtr);
MOVE.L D6,-(SP) ;Get Dialog Pointer To Close
_CloseDialog ;Close Window

; first, check our flag to make sure that driver is open

TST.W speechOK(A5)
BEQ @4 ; driver not valid

; branch around speech stuff
; driver valid, go ahead and close it

; PROCEDURE SpeechOff(theSpeech: SpeechHandle)
MOVE.L theSpeech(A5),-(SP) ; handle to speech globals
JSR SpeechOff ; close it up

@4 ; branch to here to avoid closing invalid driver

_ExitToShell ;Return To Finder

;----- Initialize Managers Subroutine -----
InitManagers
;PROCEDUREInitGraf (globalIPtr: QDPtr);
PEA -4(A5) ;Space Created For Quickdraw's Use
_InitGraf ;Init Quickdraw
_InitFonts ;Init Font Manager
_InitWindows ;Init Window Manager
;PROCEDUREInitDialogs (restartProc: ProcPtr);
CLR.L -(SP) ; NIL restart proc
_InitDialogs ;Init Dialog Manager
;procedure TEinit
_TEInit
_InitCursor ; set arrow cursor
RTS ; end of InitManagers

;----- Static Data -----

NULL DC.W 0 ; null string
```

```
/OUTPUT CheapTalkCode
```

```
; Since this code file will not run successfully until it has been  
; joined with the resources by RMaker, set its file type so  
; that it cannot be mistakenly run from the desktop.  
; Link output files are usually of type APPL
```

```
/TYPE 'CODE' 'LINK'
```

```
; link our code, CheapTalk, with the glue for the speech driver  
; routines
```

```
CheapTalk  
SpeechASM
```

```
$
```

- * CheapTalk.R
- * create the application Cheap Talk
- * First define all the resources, and then include the code
- * output file name, File type, file creator

```
MDS2:Cheap Talk  
APPLCHTK
```

- * dialog resource is a vanilla dialog
- * make it pre-loaded (4) to speed things up

```
Type DLOG  
,1 (4)
```

```
60 100 260 400  
Visible NoGoAway  
1  
0  
1
```

- * DITL resource for dialog has one static text item,
- * one edit text item,
- * and two buttons: 'Say it' and 'Quit'
- * The 'Say it' button is item #1 so that hitting return is
- * the same as clicking 'Say it'
- * make it pre-loaded (4) to speed things up

```
Type DITL  
demo,1 (4)  
4
```

```
Button  
170 200 190 250  
Say it
```

```
Button  
170 50 190 100  
Quit
```

```
EditText  
40 30 150 270  
Enter English text here
```

StaticText Disabled

10 30 30 290

This is a talking dialog demonstration

- * PHNM resource is defined by us to be a string without length
- * byte it is a phonetic translation of the static text in the DITL
- * of the same resource #
- * make it pre-loaded (4) to speed things up

Type PHNM = GNRL

demo,1 (4)

.S

DHIH9S, IHZ AH TAO4KIHNNX DAY6AELAA1G

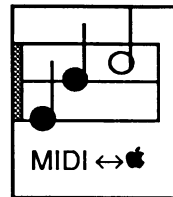
DIH1MUNSTREY5SHUN #

- * now include the code produced by the linker
- ```
INCLUDE MDS2:CheapTalkCode
```



# Sound Lab

## The Midi Connection, Part II



Kirk Austin  
MacTutor Vol. 1 No. 12

As you may recall from last issue all we need to get MIDI up and running on the Macintosh, now that we have the necessary hardware, are the driver routines for the serial ports. Fear not fellow coders, the coveted routines have arrived.

The normal device driver model for the Macintosh has too much overhead associated with it to make it appropriate for use with MIDI, which transfers data at a rate of 31.25K bits per second. Usually, real time applications need all the extra time they can get. This is particularly true of sequencer type applications that do MIDI "multi-track recording" while simultaneously maintaining a graphics display of some sort. Routines that directly access the 8530 SCC chip have been utilized in order to minimize the time taken up by the serial I/O. I have tried to make things as easy as possible by providing "building block" style routines that follow the guidelines of the Lisa Pascal interface. This is the reason for the LINK and UNLNK instructions, which aren't really necessary otherwise. You can treat RxMIDI as a Pascal function that accepts no arguments and returns a word of data as a result that's either a valid MIDI byte, or else as a flag indicating that no MIDI data is available. TxMIDI can be treated as a Pascal procedure that accepts a word of data containing the MIDI byte to be transmitted as an argument. Both of these routines are stack based.

Since these routines include interrupt handlers with pointers placed in low memory the routines cannot be in a relocatable block. One easy way of insuring this is to place them in your first code segment. If your code only consists of one segment you don't have anything to worry about, but if there is more than one segment the memory manager may move things around on you when you don't expect it. If your interrupt handlers are in one of these relocatable segments the pointers to them can be invalidated, which would cause the program to crash.

Okay, let's get down to it. First the SCC chip has to be initialized by calling either SCCinitA or SCCinitB (this should be done when your application initializes quickdraw and the various managers). The initialization ends up being quite a bit of code actually, and if not done properly will mess things up but good! The 8530 can only be accessed at a maximum rate of every 2.2 microseconds. I know, this sounds unbelievable for a sophisticated piece of modern hardware, but it's true. This is the reason for all of the MOVE.L (SP),(SP) instructions. They don't accomplish anything, but they take a little more than a couple of

microseconds to execute which is just the amount of delay that we need (silly huh?). As I mentioned in the article on the hardware interface, the SCC chip can accept three different external clock frequencies to produce the desired baud rate. The appropriate divisor must be selected in the initialize routine. By the way, these routines are written so you can use either the modem port or the printer port, but if you want to use both simultaneously and keep the ports completely independent, you will have to duplicate everything for each port. If you are writing an application of that complexity, I think you can handle rewriting these routines.

The transmit and receive routines each maintain circular queues of outgoing and incoming data respectively. These queues can be of any length, but I have arbitrarily set them at \$100 bytes each. To change the size of the queues just change the values in the equate table. There is no error detection code for a queue overrun condition, so you will have to make sure that your application runs fast enough to avoid this. If an overrun condition occurs you will lose data. If you want to dump huge files all at once (if you are writing a patch librarian for example) just make the queue big enough to handle the entire file and you can do it without worrying about overruns.

The TxMIDI routine is the most complicated one, so let's have a look at it. When this routine receives a byte to transmit (in the lower byte of the word left on the stack by the calling routine) it first must check the queue to see if it is empty or not. If the queue is not empty, the byte is simply added to the queue. If the queue is empty, the routine checks the SCC chip to see if its transmit buffer is empty. If the transmit buffer is not empty, the byte is just added to the queue. But if the transmit buffer is empty, the routine must write the byte to the SCC chip to transmit it.

The RxMIDI routine is more straightforward. It checks to see if there is any data in the queue, and if there is it returns it on the stack (the space for the result must be allocated by the calling routine). If there is no data available the routine returns \$FFFF as its result.

The interrupt handlers do exactly what you might expect them to. When a byte is received by the SCC chip, a receive interrupt is generated which calls the RxIntHand routine. This routine simply takes the byte from the SCC register and places it in the receive queue. The TxIntHand routine is called when the SCC chip's transmit buffer is empty. It takes a byte from the transmit queue if one is available and writes it to the SCC register. If no data is available, it clears the interrupt and

returns. One thing about the interrupt handlers that was not obvious to me when I was first coding them was the fact that register A5 must be saved at the beginning of the routines, and its value loaded from the system variable CurrentA5 in order to insure that it is pointing to the application's globals area.

Only one thing left to do, and that is reset the 8530 before you quit the application or try to print (if you are using the printer port for MIDI too). Just call SCCResetA or SCCResetB and your application will exit gracefully.

; MIDI PORT ROUTINES  
; copyright Kirk Austin 1985

; Transmit and Receive queue length equates

TxQSize EQU \$100  
RxQSize EQU \$100

; Serial Chip Addresses, offsets, and system equates

sccRBase EQU \$9FFFF8  
sccWBase EQU \$BFFFF9  
Lvl2DT EQU \$1B2  
aData EQU 6  
aCtl EQU 2  
bData EQU 4  
bCtl EQU 0  
TBE EQU 2  
CurrentA5 EQU \$904

; This is an example of how to use the routines.  
; If this were placed in your event loop your application would  
; receive MIDI data and echo it back out.

```
;MIDIThru
; CLR -(SP) ; clear space for result
; BSR RxMIDI ; fetch data
; MOVE (SP)+,D0
; CMPI #$FFFF,D0 ; any bytes available?
; BEQ NoMIDI ; if not, exit
; MOVE D0,-(SP) ; if so, transmit them
; BSR TxMIDI
; BRA MIDIThru ; check for more
```

;NoMIDI

; This section contains the necessary routines for MIDI  
;  
; These are the initialization routines which should be called  
; when you initialize quickdraw and the various managers.

; Call SCCInitA to use the modem port, and SCCInitB to use  
; the printer port.

SCCInitA

```
MOVE #aCtl,CtlOffset(A5) ; set up globals for Chn A
MOVE #aData,DataOffset(A5)
MOVE.B #%10000000,ChnReset(A5)
MOVE #24,RxIntOffset(A5)
MOVE #16,TxIntOffset(A5)
MOVE #28,SpecRecCond(A5)
BRA SCCInit
```

SCCInitB

```
MOVE #bCtl,CtlOffset(A5) ; set up globals for Chn B
MOVE #bData,DataOffset(A5)
MOVE.B #%01000000,ChnReset(A5)
MOVE #8,RxIntOffset(A5)
MOVE #0,TxIntOffset(A5)
MOVE #12,SpecRecCond(A5)
```

SCCInit

```
MOVE SR,-(SP) ; Save interrupts
MOVEM.L D0/A0-A1,-(SP) ; Save registers
ORI #$0300,SR ; Disable interrupts

MOVE.L #sccRBase,A1 ; Get base Read address
ADD CtlOffset(A5),A1 ; Add offset for control
MOVE.B (A1),D0 ; Dummy read
MOVE.L (SP),(SP) ; Delay
MOVE.L #sccWBase,A0 ; Get base Write address
ADD CtlOffset(A5),A0 ; Add offset for control
MOVE.B #9,(A0) ; pointer for SCC reg 9
MOVE.L (SP),(SP) ; Delay
MOVE.B ChnReset(A5),(A0) ; Reset channel
MOVE.L (SP),(SP) ; Delay
MOVE.B #4,(A0) ; pointer for SCC reg 4
MOVE.L (SP),(SP) ; Delay
```

; This is where you determine the external clock rate

; %01000100 = 500K  
; %10000100 = 1 Meg  
; %11000100 = 2 Meg

```
MOVE.B #%01000100,(A0) ; 16x clock, 1 stop bit
MOVE.L (SP),(SP) ; Delay
MOVE.B #1,(A0) ; pointer for SCC reg 1
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00000000,(A0) ; No W/Req
MOVE.L (SP),(SP) ; Delay
MOVE.B #3,(A0) ; pointer for SCC reg 3
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00000000,(A0) ; Turn off Rx
MOVE.L (SP),(SP) ; Delay
MOVE.B #5,(A0) ; pointer for SCC reg 5
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00000000,(A0) ; Turn off Tx
MOVE.L (SP),(SP) ; Delay
MOVE.B #11,(A0) ; pointer for SCC reg 11
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00101000,(A0) ; Make TRxC clock source
MOVE.L (SP),(SP) ; Delay
```

```

MOVE.B #14,(A0) ; pointer for SCC reg 14
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00000000,(A0) ; Disable BRGen
MOVE.L (SP),(SP) ; Delay
MOVE.B #3,(A0) ; pointer for SCC reg 3
MOVE.L (SP),(SP) ; Delay
MOVE.B #%11000001,(A0) ; Enable Rx
MOVE.L (SP),(SP) ; Delay
MOVE.B #5,(A0) ; pointer for SCC reg 5
MOVE.L (SP),(SP) ; Delay
MOVE.B #%01101010,(A0) ; Enable Tx and drivers
MOVE.L (SP),(SP) ; Delay
MOVE.B #15,(A0) ; pointer for SCC reg 15
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00001000,(A0) ; Enable DCD int for
 ; mouse
MOVE.L (SP),(SP) ; Delay
MOVE.B #0,(A0) ; pointer for SCC reg 0
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00010000,(A0) ; Reset EXT/STATUS
MOVE.L (SP),(SP) ; Delay
MOVE.B #0,(A0) ; pointer for SCC reg 0
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00010000,(A0) ; Reset EXT/STATUS
MOVE.L (SP),(SP) ; Delay
MOVE.B #1,(A0) ; pointer for SCC reg 1
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00010011,(A0) ; Enable interrupts
MOVE.L (SP),(SP) ; Delay
MOVE.B #9,(A0) ; pointer for SCC reg 9
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00001010,(A0) ; Set master int enable
MOVE.L (SP),(SP) ; Delay

MOVE.L #Lvl2DT,A0 ; get dispatch table
 ; pointer
MOVE RxIntOffset(A5),D0 ; get offset to Rx vector
LEA RxIntHand,A1 ; set Rx vector
MOVE.L A1,0(A0,D0)
MOVE TxIntOffset(A5),D0 ; get offset to Tx vector
LEA TxIntHand,A1 ; set Tx vector
MOVE.L A1,0(A0,D0)
MOVE SpecRecCond(A5),D0 ; get offset to
 ; Special vector

LEA Stub,A1
MOVE.L A1,0(A0,D0)

CLR RxByteIn(A5) ; init flags & pointers
CLR RxByteOut(A5)
MOVE.B #$FF,RxQEmpty(A5)
CLR TxByteIn(A5)
CLR TxByteOut(A5)
MOVE.B #$FF,TxQEmpty(A5)

MOVEM.L (SP)+,D0/A0-A1 ; Restore registers
MOVE (SP)+,SR ; Restore interrupts
RTS ; and return

```

; This is the routine to transmit a MIDI byte of data. To use this  
; place the byte to be transmitted as the lower 8 bits of a word  
; routine on the stack, then BSR to TxMIDI.

```

TxMIDI
LINK A6,#0 ; set frame pointer
MOVE SR,-(SP) ; Save interrupts
MOVEM.L D0/A0-A2,-(SP) ; Save registers
ORI #$0300,SR ; Disable interrupts

TST.B TxQEmpty(A5) ; is TxQueue empty?
BNE TxQE ; if so branch
MOVE TxByteIn(A5),D0 ; if not add byte to queue
LEA TxQueue(A5),A2 ; point to queue
MOVE.B 9(A6),0(A2,D0) ; place byte in queue
ADDQ #1,D0 ; update TxByteIn
CMP #TxQSize,D0
BNE @1
MOVE #0,D0
@1 MOVE D0,TxByteIn(A5)
BRA TxExit ; and exit

TxQE
MOVE.L #sccRbase,A0 ; get SCC Read Address
MOVE.L #sccWbase,A1 ; get SCC Write address
MOVE CtlOffset(A5),D0 ; get index for Ctl
BTST.B #TBE,0(A0,D0) ; transmit buffer empty?
BNE FirstByte ; if so branch
MOVE TxByteIn(A5),D0 ; if not add to queue
LEA TxQueue(A5),A2 ; point to queue
MOVE.B 9(A6),0(A2,D0) ; place byte in queue
ADDQ #1,D0 ; update index
CMP #TxQSize,D0
BNE @1
MOVE #0,D0
@1 MOVE D0,TxByteIn(A5)
MOVE.B #0,TxQEmpty(A5) ; reset queue empty flag
BRA TxExit ; and exit

FirstByte
MOVE DataOffset(A5),D0 ; get index to data
MOVE.L (SP),(SP) ; delay
MOVE.B 9(A6),0(A1,D0) ; write data to SCC
MOVE.L (SP),(SP) ; Delay

TxExit
MOVEM.L (SP)+,D0/A0-A2 ; Restore registers
MOVE (SP)+,SR ; Restore interrupts
UNLK A6 ; release frame pointer
MOVE.L (SP)+,A1 ; save return address
ADD.L #2,SP ; move past data word
MOVE.L A1,-(SP) ; put address back on stack
RTS ; and return

```

; This is the routine to receive a byte of MIDI data. To use this  
; routine treat it like a Pascal function. Leave space on the  
; stack for a word of data before BSR'ing to this routine. If the  
; routine executes is \$FFFF there was no MIDI data available.  
; If the upper byte is clear then a valid MIDI byte is in the lower  
; 8 bits.

```

RxMIDI
LINK A6,#0 ; set frame pointer
MOVE SR,-(SP) ; Save interrupts
MOVEM.L D0-D1/A0-A2,-(SP) ; Save registers

```

```

ORI #$0300,SR ; disable interrupts

TST.B RxQEmpty(A5) ; any data available?
BEQ @1 ; if so, branch
MOVE $FFFF,8(A6) ; if not, return with $FFFF
BRA RxExit

@1 MOVE RxByteOut(A5),D0 ; get index to byte out
LEA RxQueue(A5),A2 ; point to queue
MOVE.L #0,D1 ; clear data register
MOVE.B 0(A2,D0),D1 ; get MIDI data
MOVE D1,8(A6) ; place it on stack for
return
ADDQ #1,D0 ; update index
CMP #RxQSize,D0
BNE @2
MOVE #0,D0
@2 MOVE D0,RxByteOut(A5)
MOVE RxByteIn(A5),D1
CMP D0,D1 ; is queue empty?
BNE RxExit ; if not exit
MOVE.B #$FF,RxQEmpty(A5) ; if empty, set flag

RxExit
MOVEM.L (SP)+,D0-D1/A0-A2 ; Restore registers
MOVE (SP)+,SR ; restore interrupts
UNLK A6
RTS ; and return

```

; This is the interrupt routine for receiving a byte of MIDI data.  
; It places the received byte in a circular queue to be  
; accessed later by the application.

```

RxIntHand
ORI #$0300,SR ; disable interrupts
MOVEM.L D0-D1/A0-A2/A5,-(SP) ; save registers
MOVE.L CurrentA5,A5 ; make sure A5 is correct
MOVE.L #sccRBase,A0 ; get SCC address
MOVE.L #sccWBase,A1

MOVE DataOffset(A5),D0 ; get data offset
MOVE.B 0(A0,D0),D1 ; read data from SCC
MOVE.L (SP),(SP) ; Delay
LEA RxQueue(A5),A2 ; point to queue
MOVE RxByteIn(A5),D0 ; get offset to next cell
MOVE.B D1,0(A2,D0) ; put byte in queue
MOVE.B #0,RxQEmpty(A5) ; reset queue empty flag
ADDQ #1,D0 ; update index
CMP #RxQSize,D0
BNE @1
MOVE #0,D0
@1 MOVE D0,RxByteIn(A5)

MOVEM.L (SP)+,D0-D1/A0-A2/A5 ; restore registers
ANDI #$F8FF,SR ; enable interrupts
RTS ; and return

```

; This is the interrupt routine for transmitting a byte of MIDI  
; data. It checks to see if there is any data to send. If there is  
; it sends it to the SCC. If there isn't it resets the TBE interrupt  
; in the SCC and exits.

```

TxIntHand
ORI #$0300,SR ; disable interrupts
MOVEM.L D0-D1/A0-A2/A5,-(SP) ; save registers
MOVE.L CurrentA5,A5
MOVE.L #sccRBase,A0 ; get SCC address
MOVE.L #sccWBase,A1

TST.B TxQEmpty(A5) ; Is queue empty?
BEQ @1 ; if not branch
MOVE CtlOffset(A5),D0 ; get offset for control
MOVE.B #$28,0(A1,D0) ; if so, reset TBE
 ; interrupt
MOVE.L (SP),(SP) ; Delay
BRA TxIExit ; and exit
@1 MOVE TxByteOut(A5),D0 ; get index to next data
 ; byte
LEA TxQueue(A5),A2 ; point to queue
MOVE DataOffset(A5),D1 ; get data offset
MOVE.B 0(A2,D0),0(A1,D1) ; write data to SCC
MOVE.L (SP),(SP) ; Delay
ADDQ #1,D0 ; update index
CMP #TxQSize,D0
BNE @2
MOVE #0,D0
@2 MOVE D0,TxByteOut(A5)
MOVE TxByteIn(A5),D1
CMP D0,D1 ; is TxQueue empty?
BNE TxIExit ; if not exit
MOVE.B #$FF,TxQEmpty(A5) ; if empty set flag

```

```

TxIExit
MOVEM.L (SP)+,D0-D1/A0-A2/A5 ; restore registers
ANDI #$F8FF,SR ; enable interrupts
RTS ; and return

```

; This routine must be called when the application quits or the  
; system will crash due to the interrupt handling pointers  
; becoming invalid.

```

SCCResetA
MOVE #aCtl,CtlOffset(A5) ; set up globals for Chn A
MOVE #aData,DataOffset(A5)
MOVE.B #%10000000,ChnReset(A5)
MOVE #24,RxIntOffset(A5)
MOVE #16,TxIntOffset(A5)
MOVE #28,SpecRecCond(A5)
BRA SCCReset

```

```

SCCResetB
MOVE #bCtl,CtlOffset(A5) ; set up globals for Chn B
MOVE #bData,DataOffset(A5)
MOVE.B #%01000000,ChnReset(A5)
MOVE #8,RxIntOffset(A5)
MOVE #0,TxIntOffset(A5)
MOVE #12,SpecRecCond(A5)

```

```

SCCReset
MOVE SR,-(SP) ; Save interrupts
MOVE.L A0,-(SP) ; Save register
ORI #$0300,SR ; Disable interrupts

```



```

MOVE.L #sccWBase,A0 ; Get base Write address
ADD CtlOffset(A5),A0 ; Add offset for control
MOVE.B #9,(A0) ; pointer for SCC reg 9
MOVE.L (SP),(SP) ; Delay
MOVE.B ChnReset(A5),(A0) ; Reset channel
MOVE.L (SP),(SP) ; Delay
MOVE.B #15,(A0) ; pointer for SCC reg 15
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00001000,(A0) ; Enable DCD int
MOVE.L (SP),(SP) ; Delay
MOVE.B #0,(A0) ; pointer for SCC reg 0
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00010000,(A0) ; Reset EXT/STATUS
MOVE.L (SP),(SP) ; Delay
MOVE.B #0,(A0) ; pointer for SCC reg 0
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00010000,(A0) ; Reset EXT/STATUS
MOVE.L (SP),(SP) ; Delay
MOVE.B #1,(A0) ; pointer for SCC reg 1
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00000001,(A0) ; Enable mouse
; interrupts
MOVE.L (SP),(SP) ; Delay
MOVE.B #9,(A0) ; pointer for SCC reg 9
MOVE.L (SP),(SP) ; Delay
MOVE.B #%00001010,(A0) ; Set master int enable
MOVE.L (SP),(SP) ; Delay

```

```

MOVE.L (SP)+,A0 ; Restore register
MOVE (SP)+,SR ; Restore interrupts
RTS ; and return

```

; this is the space for a special condition interrupt routine

```

Stub
RTS

```

-----MIDI Globals-----

|             |      |         |                              |
|-------------|------|---------|------------------------------|
| CtlOffset   | DS.W | 1       | ; offset for channel control |
| DataOffset  | DS.W | 1       | ; offset for channel data    |
| ChnReset    | DS.B | 1       | ; SCC channel reset select   |
| RxIntOffset | DS.W | 1       | ; offset for dispatch table  |
| TxIntOffset | DS.W | 1       | ; offset for dispatch table  |
| SpecRecCond | DS.W | 1       | ; offset for dispatch table  |
| TxQueue     | DS.B | TxQSize | ; transmitted data queue     |
| TxQEmpty    | DS.B | 1       | ; Transmit queue empty flag  |
| TxByteIn    | DS.W | 1       | ; index to next cell in      |
| TxByteOut   | DS.W | 1       | ; index to next cell out     |
| RxQueue     | DS.B | RxQSize | ; received data queue        |
| RxQEmpty    | DS.B | 1       | ; receive queue empty flag   |
| RxByteIn    | DS.W | 1       | ; index to next cell in      |
| RxByteOut   | DS.W | 1       | ; index to next cell out     |

End



# Ask Prof. Mac

## HFS File Structure Explained



Steve Brecher  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 12

### Anonymous Questions

**Q.** I know you like to give credit to readers who send questions. But if I prefer that you not identify me, will you honor my request?

**A.** Yes. In particular, don't be afraid to ask "dumb" questions!

### DS versus DC

**Q.** Loftus Becker of Hartford, CT asks how one should choose between DS and DC for data storage in assembly-language programs. Is it a question of style, or are there reasons of substance to use sometimes one, sometimes the other?

**A.** The MDS Assembler DS directive reserves global variable storage that the program will address with an offset from the A5 register. The linker calculates the offset of each variable declared with DS, and substitutes the offset where you use the symbolic name of the variable. The linker also puts the total size of the global area into the CODE 0 resource, so that the Segment Loader can allocate sufficient space "below A5" when the application is started. DS stands for "define storage."

The DC directive allocates static space within your code segment, at the place where the directive appears. The assembler will generate the value(s) given as arguments to DC, which stands for "define constant." DC provides a way to assemble arbitrary data instead of instructions. The assembler generates PC-relative addressing for references to DC data.

DS global storage is not initialized when the application is loaded; DC data is initialized.

DS variables may be directly written with a MOVE instruction, e.g.,

```
Move #1234,MyDSVar(A5)
```

DC storage cannot be the direct destination of a MOVE instruction, because the 68000 does not allow the PC-relative addressing mode for the destination of a MOVE. To alter DC data, its address must first be placed in a register, and then it can be written to using register indirect addressing, e.g.,

```
Lea MyDCData,A0
Move #1234,(A0)
```

DS storage is always resident while your application is running. DC data is local to the segment in which it is

assembled, and goes away if the segment is purged.

DS storage does not use disk space within your application file; DC storage does take up disk space.

In sum, DS is best used for program variables, while DC is best used for constants.

### SFGetFile Unmounts

**Q.** I'm writing a desk accessory that shows information about all online volumes. It also allows the user to change file attributes. I've noticed that sometimes when I put up an SFGetFile dialog box to allow the user to choose a file, after SFGetFile returns my DA does not show all the volumes that were there (in the VCB queue) prior to the SFGetFile. Why?

**A.** One of the last things SFGetFile does after the user clicks Open or Cancel is to unmount any online volume that is not in a drive. I'd guess the reason it does this is to avoid cluttering the VCB queue (list of online volumes) with disks that the user inserted and then ejected while the SFGetFile dialog box was up. A user searching for a file could possibly insert and eject a large number of disks. SFGetFile's volume purge routine does not discriminate among offline volumes which were mounted previously and those which were mounted while its dialog was active.

### PREC's: Correction

In an answer last month, the format of a PREC resource was given incorrectly. A PREC resource specifies a selection of printer paper sizes. The correct format is:

|                            |                                                         |
|----------------------------|---------------------------------------------------------|
| n [1-word count] --        | number of active paper sizes                            |
| 6 pairs of words --        | vertical, horizontal paper size in<br>120ths of an inch |
| 6 Pascal-format strings -- | names of paper sizes                                    |

The first word of the resource is a count ranging up to 6 of the number of paper sizes that are "active," i.e., meaningful. The first "count" pairs of size words and the first "count" names are "active"; the remaining pairs of size words and name fields, if any, are unused.

### SystemTask within DA

**Q.** If some code in a Desk Accessory is time-consuming, can a call to SystemTask be placed in it? I have tried doing

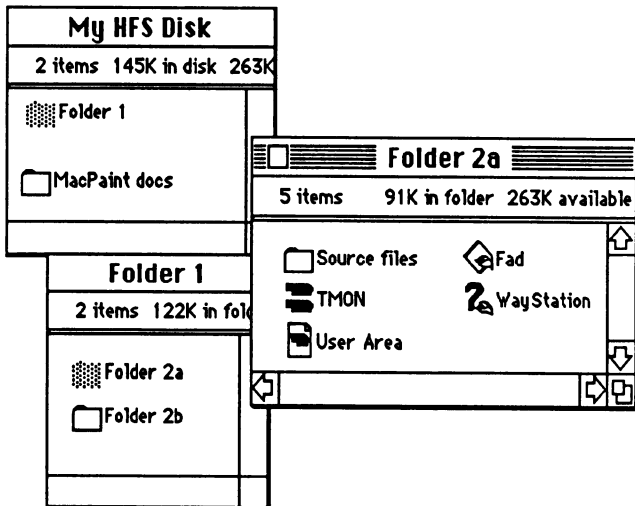


Figure 1

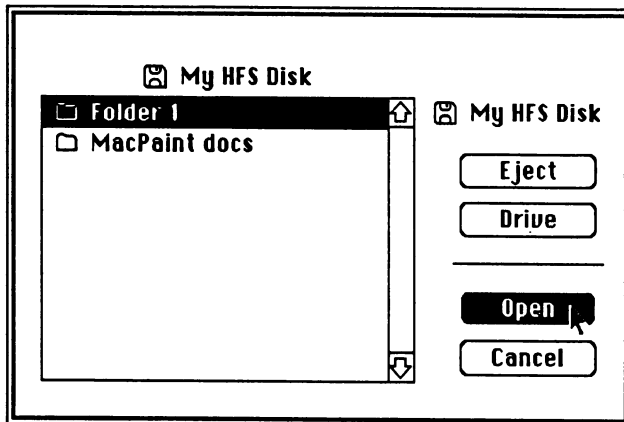


Figure 2

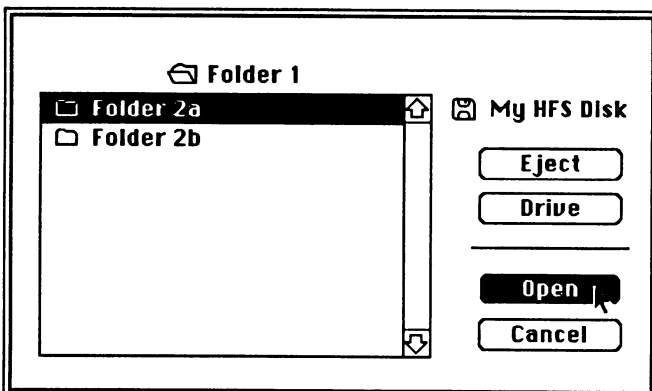


Figure 3

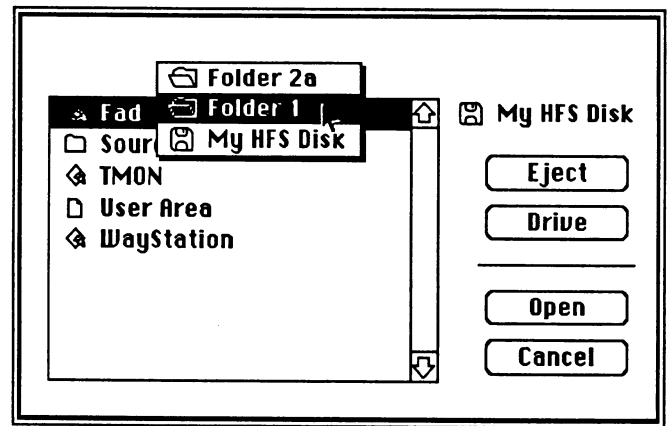


Figure 4

this with no apparent harm, but I'm concerned about re-entrancy problems.

**A.** SystemTask is a routine generally called in the main event loop of applications as a "courtesy" to desk accessories and device drivers that need to have their Control routines entered at periodic intervals. A DA or driver indicates such a need by setting the dNeedTime bit in the flags word in its header.

SystemTask uses a semaphore variable in low memory to enable it to ignore recursive calls to itself. Also, it will not generate a Control call to a driver that is currently executing. So calling SystemTask from within a DA or driver should be OK, and you needn't worry about re-entrancy because of it.

## HFS Preview

We temporarily interrupt our Q&A format to bring you a preview of HFS, the new Hierarchical File System that is being delivered with Apple's 20MB disk. Third-party disk vendors either have already implemented or are working on HFS compatibility.

HFS -- known as TFS (Turbo File System) prior to release -- replaces MFS, the original Macintosh File System. It is, in effect, a new File Manager. Along with it comes a new Finder. However, Apple has gone to great pains to make the new system upwardly compatible with existing applications and existing disks. All MFS file system calls are supported, and HFS can handle MFS disks.

HFS implements true nested subdirectories in the file system itself. Under MFS, a pseudo-subdirectory scheme was simulated by Finder folders -- but only within Finder, and at great expense in Finder execution time. Under HFS, the Finder's desktop looks much the same, but now the folders are *real* subdirectories. The new Finder does have an additional cosmetic command in the View menu, "by Small Icon," that causes files to be identified by reduced-size icons with the

filename to the right of each (see Figure 1). This option permits many more files to fit within a given Finder window; it's available for MFS disks as well as HFS disks.

With HFS, it's no longer necessary to partition hard disks into pseudo-volume subsets. Instead, users can arrange their files into folders (directories) so that many files - thousands -- can co-exist manageably on one disk volume. Directories can contain directories as well as files, making for a tree file-catalog structure. Two or more files on a disk can have the same name, as long as each is in a different directory.

HFS also uses a different scheme to allocate and keep track of used/unused space on a volume so that the minimum file allocation size can be smaller -- 0.5K, as opposed to 1K for MFS floppies or 2K-4K or more for MFS hard disks. Thus more small files can fit on a disk.

The biggest change the user will see is in the new SGetFile and SPutFile dialogs. Figures 2-4 show typical SGetFile displays of the same disk whose Finder view is shown in Figure 1. In Figure 2, the user sees the folders (and files, but there are none in the example) of the volume's root directory. He opens Folder 1, and then sees the dialog box shown in Figure 3. Folder 1 in this example also has only folders within it, although it could contain files. The user opens Folder 2a, and sees the dialog shown in Figure 4 which displays the four files and one folder that are the contents of Folder 2a. The user still hasn't located his file, and he wants to go back up the directory tree. He clicks on "Folder 2a" on top of the scroll box, and gets a pull down menu of all the directories he has traversed back through and including the root (which has the same name as the volume). By dragging the mouse down and releasing, just as for a normal menu, the user can go back to any of the higher-level directories -- he or she isn't limited to going back one level at a time.

The SPutFile dialog box (not shown) has a new scroll box-like item for entering a file name. It works

like SGetFile's, but it shows only folders (directories), not files, since its only purpose is to enable navigation among directories before specifying the name of a new file to be created.

Now let's turn to the programmer's view of HFS.

Each element of the file system catalog tree, each directory or file, is called a Catalog node (CNode). The directory at the base of the tree is called the root, and has the same name as the volume. Intermediate nodes in the tree are always directories; terminal (leaf) nodes, those with no branches extending from them, are either empty directories or files.

Each CNode has a name (such as "MacPaint Documents Folder" or "Letter to Birks, Druffey, & Co."). CNode names are strung together, with colons in between, to form a **pathname**. Each element of a pathname except the last must be a directory; the last may be either a directory or a file. A full pathname starts with the root directory name, and describes an arbitrary path to a given CNode; for example:

```
My HFS Disk:Folder 1:Folder
2a:Sources:Foo
```

A partial pathname is one that either starts with a colon or that consists entirely of a single name, e.g.,

```
:Folder 2a:Sources:Foo
Foo
```

An empty pathname element -- a pair of colons -- signifies a move up the tree, i.e., it stands for the parent of the current location. Example:

```
My HFS Disk:Folder 1:Folder
2b::Folder 2a:Sources
```

Three colons signifies a move up two levels, four colons up three levels, etc.

Internally, each directory CNode has an identifying number -- a **DirID**. CNodes can be identified to HFS system routines by using a partial pathname, together with a DirID that denotes a directory from which the partial

pathname specifies a path to the desired CNode. Note that full pathnames may in some cases exceed 255 characters in length and thus might not be representable as Pascal-format strings.

By using the new file system call **OpenWD**, the programmer can specify a **working directory**. The call returns a **WRefNum** which can be used in subsequent calls to identify a CNode. WRefNums are negative, like VRefNums, but their values won't overlap those of VRefNums. When a WRefNum is passed to a file system routine in place of a VRefNum, the system will look up the volume and the directory from the associated Working Directory Control Block (WDCB) that was filled in by an OpenWD call. The system creates a fixed pool of WDCBs at startup, and maintains its contents; the programmer need not be concerned with WDCBs directly.

The working directory scheme is one of the keys to compatibility with MFS. Under HFS, SGetFile might return a WRefNum in its reply record in the field in which it returns a VRefNum under MFS. If the program then passes that value to an **\_Open** call, the desired file will be opened; the program needn't be concerned about nor know whether SGetFile returned a WRefNum or a VRefNum.

Existing programs that pass **volume:filename** strings to the File Manager rather than using VRefNums will not be able to access HFS files (except those that are in the root directory). This includes programs written in C which use stdio routines, since those routines take only a string as a file identifier. (Specifically, it includes the programs in the MDS package.) Such programs can be fixed by inserting a call to **SetVol** between the call to SGetFile and the call to **fopen()**. If SetVol is passed a WRefNum, it will set the default directory as well as volume. If permanently changing the default volume/directory is not desirable, the previous default can be obtained first via **GetVol**, and restored after the **fopen()** with another SetVol. C stdio programs written in this way will work under both MFS and HFS.

Memory structures such as the Volume Control Block (VCB) and File Control Block (FCB) have expanded. However, all the new information is in the expansion, so that programs which access these structures under MFS will still find the old fields in the same place. Existing programs that access the FCB buffer, which is an array rather than a linked list, won't work (however Apple has been warning developers of this for some time).

As mentioned, the existing MFS File Manager calls are all supported under HFS. In addition, there are some new File Manager routines. Some of the new routines are extensions of the old ones which return additional information -- HFS-specific fields -- in expanded I/O parameter blocks. Other new routines are similar to their MFS counterparts except they accept an additional DirID field in the parameter block. The rest of the new routines have no MFS counterparts; e.g., a routine to move a file or directory from one directory to

another on the same disk (does not physically move the file, only changes the catalog); and the routines which open, close, and interrogate the contents of WDCBs. The File Manager changes will be documented in a new chapter of *Inside Macintosh* to be available soon (if not already available by the time you read this).

The initial release of HFS is RAM-resident in high memory. The final release is expected to reside in the new ROM (Rumored-Only Memory). HFS takes up additional system heap space (e.g., for the WDCB pool and an enlarged trap dispatch table); and the initial RAM version uses even more system heap space for system patches. The system heap is larger, but even so it's relatively cramped; system heap hogs beware!



# Advanced Mac'ing

## The DissBits Subroutine



Dissolve Effect

Mike Morton  
MacTutor Vol 1 No. 13

### What This Routine Does

DissBits is like copyBits: it moves one rect to another, in their respective bitMaps. It doesn't implement the modes of copyBits, nor clipping to a region. What it DOES do is copy the bits in a pseudo-random order, giving the appearance of "dissolving" from one image to another. The dissolve is rapid; the entire screen will dissolve in under four seconds.

CopyBits pays attention to the current clipping, but this routine doesn't. Other likely differences from copyBits:

- o The rectangles must have the same extents (not necessarily the same lrbt). If they are not, the routine will return -- doing nothing! No stretching copy is done as copyBits would.

- o The cursor is hidden during the dissolve, since drawing is done without quickdraw calls. The cursor reappears when the drawing is finished. For an odd effect, change it not to hide the cursor. Is this how Bill Atkinson thought of the spray can in MacPaint?

- o CopyBits may be smart enough to deal with overlapping areas of memory. This routine certainly isn't.

- o Because this routine is desperate for speed, it steals the A5 (globals) register, uses it in the central loop, then restores it before returning. If you have vertical retrace tasks which run during the dissolve, they'll wake up with the bogus A5. Since the ROM pulls this trick too, I feel this isn't a bug in MY code, but it IS more likely to expose bugs in VBL tasks. To correct the problem, have your task load A5 from the low RAM location "currentA5" immediately upon starting up, then restore its caller's A5 before returning.

You should know a few implementation details which may help:

```
;
; procedure dissBits (srcB, destB: bitMap; srcR, dstR: rect); external;
; version 5.2
;
; © by Mike Morton, September 15, 1985
; Permission to publish and distribute granted
; to MacTutor by the author. Distributed as
; "FreeWare". Contributions for using this
; routine gratefully appreciated:
; mike morton
; INFOCOM
; 125 CambridgePark Dr.
; Cambridge, MA 02140
;
; DISSBITS SUBROUTINE
; MDS ASSEMBLER VERSION
; Converted by David E. Smith for MacTutor.
;
;
; xdef dissBits
;
; Include QuickEqu.D ; MDS toolbox equates and traps
; Include SysEqu.D
; Include ToolEqu.D
; Include MacTraps.D
;
; MACRO .equ = equ| ; convert Lisa stuff to MDS
; MACRO _hidecurs = _HideCursor|
; MACRO _showcurs = _ShowCursor|
;
; definitions of the "ours" record: this structure, of which there are two copies in
; our stack frame, is a sort of bitmap:
;
; oRows .equ 0 ; (word) number of last row (first is 0)
; oCols .equ oRows+2 ; (word) number of last column (first is 0)
; oLbits .equ oCols+2 ; (word) size of left margin within 1st byte
; oStride .equ oLbits+2 ; (word) stride in memory from row to row
; oBase .equ oStride+2 ; (long) base address of bitmap
; osize .equ oBase+4 ; size, in bytes, of "ours" record
;
; stack frame elements:
;
; srcOurs .equ -osize ; (osize) our view of source bits
; dstOurs .equ srcOurs-osize ; (osize) our view of target bits
; sflast .equ dstOurs ; relative address of last s.f. member
; sfsz .equ -sflast ; size of s.f. for LINK (must be EVEN!)
;
; parameter offsets from the stack frame pointer, A6:
; last parameter is above return address and old s.f.
;
; dRptr .equ 4+4 ; ^destination rectangle
; sRptr .equ dRptr+4 ; ^source rectangle
```

```

dBptr .equ sRptr+4 ; ^destination bitMap
sBptr .equ dBptr+4 ; ^source bitMap
plast .equ sBptr+4 ; address just past last parameter
psize .equ plast-dRptr ; size of parameters, in bytes
;
; entrance: set up a stack frame, save some registers, hide the cursor.
;
dissBits: ; main entry point

 link A6,#-sfsz ; set up a stack frame
 movem.l D3-D7/A2-A5,-(SP) ; save registers compiler may need
 _hidecurs ; don't let the cursor show for now
;
; convert the source and destination bitmaps and rectangles to a format we prefer.
; we won't look at these parameters after this.
;
 move.l sBptr(A6),A0 ; point to source bitMap
 move.l sRptr(A6),A1 ; and source rectangle
 lea srcOurs(A6),A2 ; and our source structure
 bsr CONVERT ; convert to our format
 move.l dBptr(A6),A0 ; point to destination bitMap
 move.l dRptr(A6),A1 ; and rectangle
 lea dstOurs(A6),A2 ; and our structure
 bsr CONVERT ; convert to our format
;
; check that the rectangles match in size.
;
 move.w srcOurs+oRows(A6),D0 ; pick up the number of rows
 cmp.w dstOurs+oRows(A6),D0 ; same number of rows?
 bne ERROR ; nope -- bag it
 move.w srcOurs+oCols(A6),D0 ; check the number of columns
 cmp.w dstOurs+oCols(A6),D0 ; same number of columns, too?
 bne ERROR ; that's a bozo no-no
;
; figure the bit-width needed to span the columns, and the rows.
;
 move.w srcOurs+oCols(A6),D0 ; get count of columns
 ext.l D0 ; make it a longword
 bsr LOG2 ; figure bit-width
 move.w D0,D1 ; set aside that result
 beq SMALL ; too small? wimp out and do it with copyBits
 move.w srcOurs+oRows(A6),D0 ; get count of rows
 ext.l D0 ; make it a longword
 bsr LOG2 ; again, find the bit-width
 tst.w D0 ; is the result zero?
 beq SMALL ; if so, our algorithm will screw up
;
; set up various constants we'll need in the in the innermost loop
;
 move.l #1,D5 ; set up...
 lsl.l D1,D5 ; ...the bit mask which is...
 sub.l #1,D5 ; ...bit-width (cols) 1's
 add.w D1,D0 ; find total bit-width (rows plus columns)
 lea TABLE,A0 ; point to the table of XOR masks
 moveq #0,D3 ; clear out D3 before we fill the low byte
 move.b 0(A0,D0),D3 ; grab the correct XOR mask in D3
;
; the table is saved compactly, since none of the masks are wider than a byte.
; we have to unpack it so the high-order bit of the D0-bit-wide field is on:
;
UNPACK: add.l D3,D3 ; shift left by one
 bpl.s UNPACK ; keep moving until the top bit that's on is aligned at

```

o Copying from a dark area (lots of 1 bits) is slower than from a light area. But just barely (a few percent).

o There is no way to use this to randomly invert a rectangle. Instead, copyBits it elsewhere, invert it, and dissBits it back into place.

o There is also no way to slow the dissolve of a small area. To do this, copy a large area in which the only difference is the area to change.

o If you fade in a solid area, you're likely to see patterns, since the random numbers are so cheesy. Don't do this fade in nifty patterns which will distract your viewers.

o Very small areas (less than 2 pixels in either dimension) are actually done with a call to the real copyBits routine, since the pseudo-random sequence generator falls apart under those conditions.

### Calling From Languages Other Than Pascal

This routine uses the standard Lisa Pascal calling sequence. To convert it to most C compilers, you'll probably just have to delete this instruction from near the end of the main routine:

```
add.l #psize,SP;unstack parameters
```

I'd be very interested in hearing about successful uses of this routine from other languages.

### Speed of the Dissolve

You need to pay attention to this section only if:

- (a) You want the dissolve to run as fast as it can OR
- (b) You do dissolves of various sizes and want them to take proportionate lengths of time.

There are 3 levels of speed; the fastest possible one is chosen for you:

(1) An ordinary dissolve will work when moving from any bitmap to any bitmap, including on the Lisa under MacWorks. This will dissolve at about

49 microseconds per pixel. A rectangle one-quarter the size of the screen will dissolve in just over two seconds. The speed per pixel will vary slightly, and will be less if your rect extents are close to but less than powers of 2.

(2) The dissolve will speed up if both the source and destination bitmaps have rowBytes fields which are powers of two. If you're copying to the screen on a Mac, the rowBytes field already satisfies this. So make your source bitmap the right width for a cheap speedup -- about 20% faster.

(3) The fanciest level is intended for copying the whole screen. It'll paint it in about 3.4 seconds (19 microseconds per pixel). Actually, painting any rectangle which is the full width of the screen will run at this speed, for what that's worth.

### Things to Think About

- o Use a dynamically-built table to avoid the multiply instruction in the general case. This may not be a great idea since not everyone can afford that much space.

- o Adapt the routine to do transfer modes. especially pattern modes, with no source. It'd run significantly faster doing a fill of just black or white! And patXor would be pretty cool too (see also the XOR idea below).

- o Consider a front-end which does clipping (just like copyBits) by allocating yet another offscreen bitmap, whose bounds are the intersection of the destination rect with the bounding box of the destination's clipping rgn. Then:

1. Copy from the destination's bitmap into the temporary bitmap.

2. Do a copyBits from the source to the temp, with the requested masking region.

3. Dissolve from the temp to the destination, thus actually dissolving that whole rect, but changing only the clipped stuff.

A nifty way to speed up things would be to XOR the destination into the source [sic!]. Then the main loop doesn't have to copy a bit; it just tests if the bit is ON in the altered source

```
rol.l D0,D3 ; the top end. now swing the top D0 bits around to be
move.l D3,D0 ; the bottom D0 bits, the mask. 1st sequence element
 ; is the mask itself.
```

; do all kinds of preparation:

```
;
move.l srcOurs+oBase(A6),D2 ; set up base pointer for our source bits
lsl.l #3,D2 ; make it into a bit address
move.l D2,A0 ; put it where the fast loop will use it
move.w srcOurs+oLbits(A6),D2 ; now pick up source left margin
ext.l D2 ; make it a longword
add.l D2,A0 ; and make A0 useful for odd routine below
move.l dstOurs+oBase(A6),D2 ; set up base pointer for target
lsl.l #3,D2 ; again, bit addressing works out faster
move.l D2,A1 ; stuff it where we want it for the loop
move.w dstOurs+oLbits(A6),D2 ; now pick up destination left margin
ext.l D2 ; make it a longword
add.l D2,A1 ; and make A1 useful, too
move.w srcOurs+oCols(A6),A2 ; pick up the often-used count of columns
move.w srcOurs+oRows(A6),D2 ; and of rows
add.w #1,D2 ; make row count one-too-high for compares
ext.l D2 ; and make it a longword
lsl.l D1,D2 ; slide it to line up w/rows part of D0
move.l D2,A4 ; and save that somewhere useful
move.w D1,D2 ; put log2(columns) in a safe place (sigh)
```

```
;
; try to reduce the amount we shift down D2. this involves:
; halving the strides as long as each is even, decrementing D2 as we go
; masking the bottom bits off D4 when we extract the row count in the loop
```

```
;
; alas, we can't always shift as little as we want. for instance, if we don't
; shift down far enough, the row count will be so high as to exceed a halfword,
; and the dread mulu instruction won't work (it eats only word operands). so,
; we have to have an extra check to take us out of the loop early.
```

```
;
move.w srcOurs+oStride(A6),D4 ; pick up source stride
move.w dstOurs+oStride(A6),D7 ; and target stride
move.w srcOurs+oRows(A6),D1 ; pick up row count for kludgy check
tst.w D2 ; how's the bitcount?
beq.s HALFDONE ; skip out if already down to zero
```

### HALFLOOP:

```
btst #0,D4 ; is this stride even?
bne.s HALFDONE ; nope -- our work here is done
btst #0,D7 ; how about this one?
bne.s HALFDONE ; have to have both even
lsl.w #1,D1 ; can we keep max row number in a halfword?
bcs.s HALFDONE ; nope -- D2 mustn't get any smaller!
lsr.w #1,D4 ; halve each stride...
lsr.w #1,D7 ; ...like this
sub.w #1,D2 ; and remember not to shift down as far
bne.s HALFLOOP ; loop unless we're down to no shift at all
```

### HALFDONE:

```
; no tacky platitudes, please
move.w D4,srcOurs+oStride(A6) ; put back source stride
move.w D7,dstOurs+oStride(A6) ; and target stride
```

```
;
; make some stuff faster to access -- use the fact that (An) is faster to access
; than d(An). this means we'll misuse our frame pointer, but don't worry -- we'll
; restore it before we use it again.
```

```
;
move.w srcOurs+oStride(A6),A5 ; make source stride faster to access, too
move.l A6,-(SP) ; save framitz pointer
```



```

move.w dstOurs+oStride(A6),A6 ; pick up destination stride
move.l #0,D6 ; we do only AND.W x,D6 -- but ADD.L D6,x
clr.w -(SP) ; reserve room for function result
bsr MULCHK ; go see if strides are powers of two
tst.w (SP)+ ; can we eliminate the horrible MULUs?
bne NOMUL ; yes! hurray!
;
; main loop: map the sequence element into rows and columns, check if
; it's in bounds and skip on if it's not, flip the appropriate bit, generate the
; next element in the sequence, and loop if the sequence isn't done.
;
; check the row bounds. note that we can check the row before extracting it from
; D0, ignoring the bits at the bottom of D0 for the columns. to get these bits
; to be ignored, we had to make A4 one-too-high before shifting it up to align it.
;
LOOP: ; here for another time around
 cmp.l A4,D0 ; is row in bounds?
 bge.s NEXT ; no: clip this
;
; map it into the column; check bounds. note that we save this check for second;
; it's a little slower because of the move and mask.
;
; chuck sagely points out that when the "bhi" at the end of the loop takes, we
; know we can ignore the above comparison. thanks, chuck. you're a great guy.
;
LOOPROW: ; here when we know the row number is OK
 move.w D0,D6 ; copy the sequence element
 and.w D5,D6 ; find just the column number
 cmp.w A2,D6 ; too far to the right? (past oCols?)
 bgt.s NEXT ; yes: skip out
 move.l D0,D4 ; we know element will be used; copy it
 sub.w D6,D4 ; remove column's bits
 lsr.l D2,D4 ; shift down to row, NOT right-justified
;
; get the source byte, and bit offset. D4 has the bit offset in rows, and
; D6 is columns.
;
 move.w A5,D1 ; get the stride per row (in bits)
 mulu D4,D1 ; stride * row; find source row's offset in bits
 add.l D6,D1 ; add in column offset (bits)
 add.l A0,D1 ; plus base of bitmap (bits [sic])
 move.b D1,D7 ; save the bottom three bits for the BTST
 lsr.l #3,D1 ; while we shift down to a word address
 move.l D1,A3 ; and save that for the test, too
 not.b D7 ; get right bit number (compute #7-D7)
;
; find the destination bit address and bit offset
;
 move.w A6,D1 ; extract cunningly hidden destination stride
 mulu D1,D4 ; stride*row number = dest row's offset in bits
 add.l D6,D4 ; add in column bit offset
 add.l A1,D4 ; and base address, also in bits
 move.b D4,D6 ; set aside the bit displacement
 lsr.l #3,D4 ; make a byte displacement
 not.b D6 ; get right bit number (compute #7-D6)
 btst D7,(A3) ; test the D7th bit of source byte
 move.l D4,A3 ; point to target byte (don't lose CC from btst)
 bne.s SETON ; if on, go set destination on
 bclr D6,(A3) ; else clear destination bit
;
; find the next sequence element. see knuth, vol ii., page 29 for sketchy details.
;

```

bitmap and, if so, TOGGLES it in the destination. (i.e., it does a srcXor operation). To repair the source bitmap, a third XOR from the destination to the source should do it [any old-timer will recognize this triple XOR as the best way to swap two registers.] The trick is making the invisible XOR run so fast that the time to do it is less than the savings in the visible part. Or perhaps the batch XORs could be done in spare-time...

- o Implement a partial dissolve; this would facilitate animation of Star Trek-style teleporting. There are a lot of ways to do this. The easiest might be to include a counter in the loop ("A register! My kingdom for a register!"). Alternately, we could assume that the list of starting and ending points could be limited, allowing a table-based system. This is something worth looking into, but could be real difficult.

- o Look at rearranging instructions so CPU-intensive ones aren't grouped, allowing us to sneak in between video cycles more often.

- o Add some real error-handling. How should we do this? Return a status? Fault? We could also return info to our caller on which of the three loops got used, so they know they're getting the speed they want.

- o Don't use the BITWIDTH routine to test for exact powers of two in MULCHK. A number X is an exact power of two if (x & -x) equals x.

I wish to retain the rights to this routine, so please respect the "FreeWare" concept and do not use it within a commercial software product without a registration card. To register, please send your name, address and intended use along with a donation to the address shown in the front of the dissbits source code.

*Why not subscribe  
to MacTutor, and get  
this kind of technical help  
every month. Call (714)  
630-3730 for more  
information.*

```

NEXT: ; jump here if D0 not in bounds
 lsr.l #1,D0 ; slide one bit to the right
 bhi.s LOOPROW ; if no carry out, but not zero, loop
 eor.l D3,D0 ; flip magic bits appropriate to the bitwidth we want...
 cmp.l D3,D0 ; ...but has this brought us to square 1?
 bne.s LOOP ; if not, loop back; else...
 bra DONE ; ...we're finished

SETON:
 bset D6,(A3) ; source bit was on: set destination on
 lsr.l #1,D0 ; slide one bit to the right
 bhi.s LOOPROW ; if no carry out, but not zero, loop
 eor.l D3,D0 ; flip magic bits...
 cmp.l D3,D0 ; ...but has this brought us to square 1?
 bne.s LOOP ; if not, loop back; else fall through

;
; here when we're done; the (0,0) point has not been done yet. this is
; really the (0,left margin) point. we also jump here from another copy loop.
;
DONE:
 move.l (SP)+,A6 ; restore stack frame pointer
 move.w srcOurs+oLbits(A6),D0 ; pick up bit offset of left margin
 move.w dstOurs+oLbits(A6),D1 ; and ditto for target
 not.b D0 ; flip to number the bits for 68000
 not.b D1 ; ditto

;
; alternate, late entrance, when SCREEN routine has already set up D0
; and D1 (it doesn't want the bit offset negated).
;
DONEA: ; land here with D0, D1 set
 move.l srcOurs+oBase(A6),A0 ; set up base pointer for our source bits
 move.l dstOurs+oBase(A6),A1 ; and pointer for target
 bset D1,(A1) ; assume source bit was on; set target
 btst D0,(A0) ; was first bit of source on?
 bne.s DONE2 ; yes: skip out
 bclr D1,(A1) ; no: oops! set it right, and fall through

;
; return
;
DONE2: ; here when we're really done
ERROR: ; we return silently on errors
 _showcurs ; let's see this again
 movem.l (SP)+,D3-D7/A2-A5 ; restore lots of registers
 unlk A6 ; restore caller's stack frame pointer
 move.l (SP)+,A0 ; pop return address
 add.l #psize,SP ; unstack parameters
 jmp (A0) ; home to mother

;
; -----
;
; sleazo code for when we're asked to dissolve very small regions.
; if either dimension of the rectangle is too small, we bag it and just
; delegate the problem to copyBits. a possible problem with this is if
; someone decides to substitute us for the standard copyBits routine
; -- this case will become recursive...
;
SMALL: ; here when it's too small to copy ourselves
 move.l sBptr(A6),-(SP) ; push args: source bitmap
 move.l dBptr(A6),-(SP) ; destination bitmap
 move.l sRptr(A6),-(SP) ; source rectangle
 move.l dRptr(A6),-(SP) ; destination rectangle
 move.w #srcCopy,-(SP) ; transfer mode -- source copy

```

```

clr.l -(SP) ; mask region -- NIL
_copyBits ; do the copy in quickdraw-land
bra DONE2 ; head for home
;
;-----
; code identical to the usual loop, but A5 and A6 have been changed to
; shift counts. other than that, it's the same. really it is! well, no, wait a minute...
; because we don't have to worry about the word-size mulu operands, we can
; collapse the shifts and countershifts further as shown below:

NOMUL: ; here for alternate version of loop
 tst.w D2 ; is right shift zero?
 beq.s NOMUL2 ; yes: can't do much more...
 cmp.w #0,A5 ; how about one left shift (for source stride)?
 beq.s NOMUL2 ; yes: ditto
 cmp.w #0,A6 ; and the other left shift (destination stride)?
 beq.s NOMUL2 ; yes: can't do much more...

 sub.w #1,D2 ; all three...
 sub.w #1,A5 ; ...are...
 sub.w #1,A6 ; ...collapsible
 bra.s NOMUL ; go see if we can go further
;
; see if we can do the super-special-case loop, which basically is equivalent to any
; rectangle where the source and destination are both exactly the width of the Mac
; screen.
;
NOMUL2: ; here when D2, A5, and A6 are all collapsed
 tst.w D2 ; did this shift get down to zero?
 bne.s NLOOP ; no: skip to first kludged loop
 cmp.w #0,A5 ; is this zero?
 bne.s NLOOP ; no: again, can't make further optimization
 cmp.w #0,A6 ; how about this?
 bne.s NLOOP ; no: the best-laid plans of mice and men...
 cmp.w A2,D5 ; is there no check on the column?
 bne.s NLOOP ; not a power-of-two columns; rats!
 move.w A0,D6 ; grab the base address of the source
 and.b #7,D6 ; select the low three bits
 bne.s NLOOP ; doesn't sit on a byte boundary; phooey
 move.w A1,D6 ; now try the base of the destination
 and.b #7,D6 ; and select its bit offset
 beq.s SCREEN ; yes! do extra-special loop!
;
; fast, but not super-fast loop, used when both source and destination bitmaps
; have strides which are powers of two.
;
NLOOP: ; here for another time around
 cmp.l A4,D0 ; is row in bounds?
 bge.s NNEXT ; no: clip this

NLOOPROW: ; here when we know the row number is OK
 move.w D0,D6 ; copy the sequence element
 and.w D5,D6 ; find just the column number
 cmp.w A2,D6 ; too far to the right? (past oCols?)
 bgt.s NNEXT ; yes: skip out
 move.l D0,D4 ; we know element will be used; copy it
 sub.w D6,D4 ; remove column's bits
 lsr.l D2,D4 ; shift down to row, NOT right-justified
 move.w A5,D7 ; get log2 of stride per row (in bits)
 move.l D4,D1 ; make a working copy of the row number
 lsl.l D7,D1 ; * stride/row is source row's offset in bits

```

```

add.l D6,D1 ; add in column offset (bits)
add.l A0,D1 ; plus base of bitmap (bits [sic])
move.b D1,D7 ; save the bottom three bits for the BTST
lsl.l #3,D1 ; while we shift down to a byte address
move.l D1,A3 ; and save that for the test, too
not.b D7 ; get right bit number (compute #7-D7)

move.w A6,D1 ; extract log2 of destination stride
lsl.l D1,D4 ; stride*row number = dest row's offset in bits
add.l D6,D4 ; add in column bit offset
add.l A1,D4 ; and base address, also in bits
move.b D4,D6 ; set aside the bit displacement
lsl.l #3,D4 ; make a byte displacement
not.b D6 ; get right bit number (compute #7-D6)
btst D7,(A3) ; test the D7th bit of source byte
move.l D4,A3 ; point to target byte (don't ruin CC from btst)
bne.s NSETON ; if on, go set destination on
bclr D6,(A3) ; else clear destination bit

```

```

NNEXT: ; jump here if D0 not in bounds
lsl.l #1,D0 ; slide one bit to the right
bhi.s NLOOPROW ; if no carry out, but not zero, loop
eor.l D3,D0 ; flip magic bits...
cmp.l D3,D0 ; ...but has this brought us to square 1?
bne.s NLOOP ; if not, loop back; else...
bra DONE ; ...we're finished

```

```

NSETON:
bset D6,(A3) ; source bit was on: set destination on
lsl.l #1,D0 ; slide one bit to the right
bhi.s NLOOPROW ; if no carry out, but not zero, loop
eor.l D3,D0 ; flip magic bits...
cmp.l D3,D0 ; ...but has this brought us to square 1?
bne.s NLOOP ; if not, loop back; else fall through
bra DONE ; and finish

```

```

;
; -----
;
; super-special case, which happens to hold for the whole mac screen -- or subsets
; of it which are as wide as the screen. here, we've found that the shift counts
; in D2, A5, and A6 can all be collapsed to zero. and D5 equals A2, so there's
; no need to check whether D6 is in limits -- or even take it out of D0! so, this loop
; is the NLOOP code without the shifts or the check on the column number. should
; run like a bat; have you ever seen a bat run?
;
; oh, yes, one further restriction -- the addresses in A0 and A1 must point to
; integral byte addresses with no bit offset. (this still holds for full-screen
; copies.) because both the source and destination are byte-aligned, we can skip
; the ritual Negation Of The Bit Offset which the 68000 usually demands.

```

```

SCREEN: ; here to set up to do the whole screen, or at least its
width
move.l A0,D6 ; take the base source address...
lsl.l #3,D6 ; ... and make it a byte address
move.l D6,A0 ; replace pointer
move.l A1,D6 ; now do the same...
lsl.l #3,D6 ; ...for...
move.l D6,A1 ; ...the destination address
bra.s N2LOOP ; jump into loop

```

```

N2HEAD:
eor.l D3,D0 ; here when we shifted and a bit carried out
 ; flip magic bits to make the sequence work

```

```

N2LOOP: ; here for another time around
 cmp.l A4,D0 ; is row in bounds?
 bge.s N2NEXT ; no: clip this

N2LOOPROW: ; here when we know the row number is OK
 move.l D0,D1 ; copy row number, shifted up, plus column offset
 lsr.l #3,D1 ; while we shift down to a word offset
 btst D0,0(A0,D1) ; test bit of source byte
 bne.s N2SETON ; if on, go set destination on
 bclr D0,0(A1,D1) ; else clear destination bit

N2NEXT: ; jump here if D0 not in bounds
 lsr.l #1,D0 ; slide one bit to the right
 bhi.s N2LOOPROW ; if no carry out, but not zero, loop
 bne.s N2HEAD ; if carry out, but not zero, loop earlier
 bra.s N2DONE ; 0 means next sequence element would have been D3

N2SETON:
 bset D0,0(A1,D1) ; source bit was on: set destination on
 lsr.l #1,D0 ; slide one bit to the right
 bhi.s N2LOOPROW ; if no carry out, but not zero, loop
 bne.s N2HEAD ; if carry out, but not zero, loop earlier
 ; zero means the loop has closed on itself

;
; because our bit-numbering isn't like that of the other two loops, we set up D0 and D1
; ourselves before joining a bit late with the common code to get the last bit.
;
N2DONE:
 move.l (SP)+,A6 ; restore the stack frame pointer
 move.w srcOurs+oLbits(A6),D0 ; pick up bit offset of left margin
 move.w dstOurs+oLbits(A6),D1 ; and ditto for target
 bra DONEA ; go do the first bit, which the sequence doesn't cover
;
; -----
;
; mulchk -- see if we can do without multiply instructions.
;
; calling sequence:
; A5 holds the source stride
; A6 holds the destination stride
; clr.w -(SP) ; reserve room for boolean function return
; bsr MULCHK ; go check things out
; tst.w (SP)+ ; test result
; bne.s SHIFT ; if non-zero, we can shift and not multiply
;
; (if we can shift, A5 and A6 have been turned into shift counts)
; registers used: none (A5, A6)

MULCHK:
 movem.l D0-D3,-(SP) ; stack caller's registers
 move.l A5,D0 ; take the source stride
 bsr BITWIDTH ; take log base 2
 move.l #1,D1 ; pick up a one...
 lsl.l D0,D1 ; ...and try to recreate the stride
 cmp.l A5,D1 ; does it come out the same?
 bne.s NOMULCHK ; nope -- bag it
 move.w D0,D3 ; save magic logarithm of source stride
 move.l A6,D0 ; yes -- now how about destination stride?
 bsr BITWIDTH ; convert that one, also
 move.l #1,D1 ; again, try a single bit...
 lsl.l D0,D1 ; ...and see if original # was 1 bit

```

```

 cmp.l A6,D1 ; how'd it come out?
 bne.s NOMULCHK ; doesn't match -- bag this
;
; we can shift instead of multiplying. change address registers & tell our caller.
;
 move.w D3,A5 ; set up shift for source stride
 move.w D0,A6 ; and for destination stride
 st 4+16(SP) ; tell our caller what's what
 bra.s MULRET ; and return

NOMULCHK:
 sf 4+16(SP) ; tell caller we can't optimize
MULRET:
 movem.l (SP)+,D0-D3 ; pop some registers
 rts ; all set
;
; -----
;
; table of (longword) masks to XOR in strange Knuthian algorithm. the first table
; entry is for a bit-width of two, so the table actually starts two bytes before
; that. hardware jocks among you may recognize this scheme as the software
; analog
; of a "maximum-length sequence generator".
;
; to save a bit of room, masks are packed in bytes, but should be aligned as
; described in the code before being used.
;
 .ALIGN 2
table: DC.B 0,0 ; first element is #2
 DC.B 3 ; 2
 DC.B 3 ; 3
 DC.B 3 ; 4
 DC.B 5 ; 5
 DC.B 3 ; 6
 DC.B 3 ; 7
 DC.B 23 ; 8
 DC.B 17 ; 9
 DC.B 9 ; 10
 DC.B 5 ; 11
 DC.B 101 ; 12
 DC.B 27 ; 13
 DC.B 53 ; 14
 DC.B 3 ; 15
 DC.B 45 ; 16
 DC.B 9 ; 17
 DC.B 129 ; 18
 DC.B 57 ; 19
 DC.B 9 ; 20
 DC.B 5 ; 21
 DC.B 3 ; 22
 DC.B 33 ; 23
 DC.B 27 ; 24
 DC.B 9 ; 25
 DC.B 113 ; 26
 DC.B 57 ; 27
 DC.B 9 ; 28
 DC.B 5 ; 29
 DC.B 101 ; 30
 DC.B 9 ; 31
 DC.B 163 ; 32
 .align 2
;

```

```

;-----
;
; convert -- convert a parameter bitMap and rectangle to our internal form.
;
; calling sequence:
; lea bitMap,A0 ; point to the bitmap
; lea rect,A1 ; and the rectangle inside it
; lea ours,A2 ; and our data structure
; bsr CONVERT ; call us
;
; when done, all fields of the "ours" structure are filled in:
; oBase is the address of the first byte in which any bits are to be changed
; oLbits is the number of bits into that first byte which are ignored
; oStride is the stride from one row to the next, in bits
; oCols is the number of columns in the rectangle
; oRows is the number of rows
;
; registers used: D0, D1, D2
;
CONVERT:
;
; save the starting word and bit address of the stuff:
;
; move.w top(A1),D0 ; pick up top of inner rectangle
; sub.w bounds+top(A0),D0 ; figure rows to skip within bitmap
; mulu rowbytes(A0),D0 ; compute bytes to skip (relative offset)
; add.l baseaddr(A0),D0 ; find absolute address of first row to use
; move.w left(A1),D1 ; pick up left coordinate of inner rect
; sub.w bounds+left(A0),D1 ; find columns to skip
; move.w D1,D2 ; copy that
; and.w #7,D2 ; compute bits to skip in first byte
; move.w D2,oLbits(A2) ; save that in the structure
; lsr.w #3,D1 ; convert column count from bits to bytes
; ext.l D1 ; convert to a long value, so we can...
; add.l D1,D0 ; add to row start in bitmap to find 1st byte
; move.l D0,oBase(A2) ; save that in the structure
;
; save the stride of the bitmap. this is the same as for the original, but in bits.
;
; move.w rowbytes(A0),D0 ; pick up the stride
; lsl.w #3,D0 ; multiply by eight to get a bit stride
; move.w D0,oStride(A2) ; stick it in the target structure
;
; save the number of rows and columns.
;
; move.w bottom(A1),D0 ; get the bottom of the rectangle
; sub.w top(A1),D0 ; less the top coordinate
; sub.w #1,D0 ; get number of highest row (1st is zero)
; bmi.s CERROR ; nothing to do? (note: 0 IS ok)
; move.w D0,oRows(A2); ; save that in the structure
; move.w right(A1),D0 ; get the right edge of the rectangle
; sub.w left(A1),D0 ; less the left coordinate
; sub.w #1,D0 ; make it zero-based
; bmi CERROR ; nothing to do here?
; move.w D0,oCols(A2) ; save that in the structure
;
; all done. return.
;
; rts
;
; error found in CONVERT. pop return and jump to the error routine, such as it is.
;

```

#### **CERROR:**

```

addq.l #4,SP; pop four return address.
bra ERROR ; return silently
;
;-----

```

```

; log2 -- find the ceiling of the log, base 2, of a
; number.

```

```

; bitwidth -- find how many bits wide a number
; is

```

```

; calling sequence:

```

```

; move.l N,D0 ; store the number in D0
; bsr LOG2 ; call us
; move.w D0,... ; D0 contains word result
;

```

```

; registers used: D2, (D0)
;

```

#### **BITWIDTH:**

```

sub.l #1,D0 ; so 2**n works right (sigh)

```

#### **LOG2:**

```

tst.l D0 ; did they pass us a zero?
beq.s LOGDONE ; call log2(0)
beq.s LOGDONE ; if D0 was one,
answer is zero

```

```

move.w #32,D2 ; initialize count

```

#### **LOG2LP:**

```

lsl.l #1,D0 ; slide bits to the left by one
dbcs D2,LOG2LP ; decrement
move.w D2,D0 ; save our value

```

```

LOGDONE: ; here with final value in D0
rts ; and return

```

```

END ; procedure dissBits

```





### Selecting a Compiler

Why the title "A New World"? Because programming on the Mac is exactly that. No matter what your background, you'll find yourself in a strange environment on the Mac. Therefore it's really important that you have a compiler that is suitable and compatible.

I am not going to review the available compilers here. Rather, I'll outline the things you should look for in a compiler and library.

First, the essentials ... the compiler must:

- 1) Produce position-independent 68000 code,
- 2) Provide some form of access to Mac system services via the "trap" mechanism,
- 3) Be compatible with a relocating linker which can combine assembler and C modules with full segment loader and resource manager compatibility, and
- 4) Not require a run-time support library.

### Position-Independent Code

Mac programs must be capable of running when loaded anywhere in memory. This imposes restrictions on the use of addressing modes and pointers and the compiler must adhere to those restrictions.

At this point, I'd like to make the distinction between *relocatable* code and *position-independent* code. The former refers to the output of a compiler or assembler which is designed to work with a linker that combines modules into a final executable program image. The linker "relocates" address references as it combines modules.

The latter, position-independent code, refers to machine code (generated by any language) which does not depend on being loaded at a specific location in physical memory. Consider the C construct:

```
static char *key[] = {"ME", "YOU" ...};
```

which generates an array of pointers to strings. This is not a position-independent construct. The pointers are locked at link time. How does the C language system handle this sort of thing on the Mac?

### Macintosh Traps

All Mac applications will make heavy use of the rich set of system services provided by the Macintosh operating system

in RAM and ROM. These have become known in Mac circles as *traps*, since the "unimplemented instruction trap" is used to pass control to the service routines in the system.

The first challenge faced by any new Mac programmer is to translate the trap descriptions found in *Inside Macintosh* from Pascal to whatever. Some hints are given for assembly programmers, none for C people.

The traps are called with arguments on the stack in a specific way dictated by the Lisa Pascal language run-time system. The C compiler must provide a useful way to access the trap services without forcing you into a Pascal-like rigidity.

An example of the kinds of issues we are dealing with here is the difference in the way strings are handled. The C

During one of my recent visits to the CompuServe "MAUG" forum, I was particularly struck by a message from a disappointed new Mac owner. His complaint: "It's the first computer I've ever heard of that I can't program."

He was, of course, not talking about MS Basic or Forth. What he meant was that he couldn't develop "real" programs on it. Things have changed. Now there are several language systems available which do support native program development, for the Mac, on the Mac.

This column, the *C Workshop*, is dedicated to providing information to those of you who are interested in native Macintosh programming. We will assume that you know the C language (this is not a C tutorial column). You will need a copy of *Inside Macintosh* as well. This 2-volume description of Macintosh internals is essential to any developer.

Bob Denny

language treats strings as free-form sequences of characters ending in a null or zero byte.

On the other hand, standard Pascal has no variable-length data capability. So the Mac developers invented a Pascal data structure called *STR255*, which consists of a 1-byte count word followed by 255 bytes of fixed-length storage. Ever wonder why Pascal programs seem to require so much memory?

Recall that the traps mimic the Pascal interface. So they expect strings in Pascal form. We certainly don't want to handle strings in Pascal form in our C programs (though we could). Some type of conversion between the C and Pascal string formats is indispensable. Does the compiler do it? Does the compiler call a run-time library function to do it transparently when you call a trap? Or is a function provided



for you, keeping things under your control?

### Linking Modules

One of the big advantages of C over Pascal is the ease with which the program can be written in separately compiled modules. When developing on the Mac, this is particularly important. You can save an enormous amount of development time on a large application by working on it in small pieces.

For this to work, the compiler must generate relocatable code compatible with a linker that is reasonably fast. There should be an assembler which is compatible with the linker as well, making it possible to combine assembly and C language modules in a single image.

The linker *must* generate a program image whose internal structure is compatible with the Mac segment loader and resource manager. When comparing C language systems, these are the questions you should ask first: "Can program images be launched by the finder?" and "Can I produce applications with resources in the program file?"

If you are used to the MS-DOS and CP/M-86 relocatable assembler/linker packages, or those found on minicom-puters, you are in for a nasty surprise. I have yet to find a linker which generates segment loader images and has selective load from a libraries. Believe it or not, the Macintosh 68000 Development System (MDS) linker does not support true libraries! There is no librarian.

### Run-Time Libraries

C programmers who are used to the UNIX environment often lose sight of a very important feature of C. The language has no built-in I/O, and no implicit operations on structured data types. The spirit of the language is "no code explosion, no run-time support required".

You should look very carefully at the run-time support requirements of the various C compilers. This can be tricky, but the time spent is well worth it.

We have already said that our compiler must generate 68000 code. But some- times this is impractical. At one extreme, the compiler might generate inline code for everything, including things not supported by the 68000 architecture. At the other extreme, the compiler might generate nothing but calls to a run-time library ("P-code" and threaded-code systems fall into this category).

For our purposes, the compiler ought to generate inline code wherever practical, and call library support routines only for things not supported by the 68000 architecture such as 32-bit multiplies and divides and floating point.

Floating-point operations require software support on the Mac. Does the compiler use the Mac's floating point support or a separate library developed for the compiler? If it does use the Mac's support, how efficiently? Does it generate inline code to set up the calls or is a "glue" routine used? Finally, do you even need floating point?

Most C language systems are supplied with a UNIX-

emulating "standard I/O" library. This makes it possible to quickly move existing C applications and software tools to the Mac. Depending on the linker and library support, the library may be a large inseparable block of code. As you get further along in native application development, you'll find that you don't want the UNIX emulation. Rather, you'll be making heavy use of the Mac's toolbox and packages such as "Standard File".

Any C system worth considering should come with both a UNIX library and a minimum support library which contains only those functions needed to support the code generated by the compiler and the interface to the Mac trap mechanism. How big is this minimum library?

### Some Desirable Things

What we have covered so far falls into the "required" class. Here are a few things that might be supplied with the C system which might make your life easier or your programs smaller and/or faster.

- 1) "H" files mapping Macintosh data structures for use with toolbox access.
- 2) Native Macintosh library, with functions easing use of windows, dialogs, standard file, etc.
- 3) Trap access without interface library, i.e., in-line trap calls.
- 4) In-line assembly language.
- 5) Library sources.

### Conclusions

Selecting a C system for use on the Macintosh is difficult. The environment is very different from any other computer and operating system you may have known. This fact might not have been appreciated by the language implementor. In fact, the system may be an ill-suited "port" from some other environment.

On the other hand, a good C system would certainly rate toward the top as a language well suited to the data structure rich Macintosh environment.

I have selected Mac C from Consulair Corporation. It integrates with the Macintosh 68000 Development System, comes with a complete set of "H" files, a UNIX library, a minimum library and a Mac-oriented library. It supports in-line assembler and generates in-line code for all Mac trap calls. It is possible to write programs in Mac C which require no library at all.

In future columns, I will present an evolving program which will serve as an example of a Mac application which adheres to the user interface guidelines and uses the Mac's toolbox traps. The program will be written in Mac C.

## Introducing the Mac Environment

At the outset, I mentioned that the Macintosh programming environment is unlike that of any computer you have ever seen. From the C programmer's view- point, there are some fundamental clashes with common C techniques.

Remember that the Mac is heavily oriented toward a Pascal environment. Pascal has no facilities for creating fixed data structures at compile time. Rather, the typical Pascal program includes an "initialization procedure" which copies the contents of fixed structures into their run-time locations. The initialization procedure often makes up a large fraction of the total program size, yet does no useful work.

The Macintosh developers realized this and created the "resource" concept. Resources are fixed blocks of structured data such as text strings, window descriptions, icon bitmaps, etc. There is a "resource maker" that creates these data structures from a special language. Resources and code are written in two separate languages. The Mac provides system services (traps) to handle resources at run time.

So a Pascal program's initialization procedure calls a system service to perform the actual initialization, copying resources from files into memory. The static structures for the program are brought together with the code at program startup.

Since resources are separately compiled and stored, and managed by the operating system, it is possible to maintain a resource separately from the program. For example, the text messages issued by an application might be changed from English to French without touching the program image.

On the minus side, resource data is accessible only by a pointer or handle (pointer to a pointer) returned by the resource manager.

Meanwhile, what about statically initialized variables in C? There are plenty of situations where the resource concept is inappropriate for static data. Are we stuck with it anyway? The Mac segment loader has no support for loading static data into the "application globals" area and, worse, static data must be position-independent. You should consult your compiler's documentation to see how it handles these things.

Some things that you may have grown accustomed to over the last few years, such as memory mapping, linker library support, exception handling and layered interfaces just aren't present in the Mac.

The scheduling architecture is unusual. You must provide support for "desk accessories" in your application, but the support consists of calling system services. The disk driver performs input services for the modem serial port even though there is a separate serial driver.

Sometimes it seems like the Mac's designers skipped the evolutionary period of the Seventies entirely, leaving a patchwork of old techniques and brand new (and wonderful) ergonomics.

In future C Workshops, we'll address these things individually. It will be difficult to get used to the Mac environ- ment, and our purpose is to ease that task by sharing our experiences and augmenting the information in Inside Macintosh.



# C Workshop

## Application Template



Bob Denny  
Editorial Board  
MacTutor Vol. 1 No. 2

This month's C Workshop presents a "template" application. Most Macintosh programs share a common overall structure, dictated by the operating system environment. The purpose of the template program is to give you a framework for developing your own applications. It will also be used as the basis for topics discussed in future C Workshops. The functions that do the actual work of the program have been "stubbed".

Since "a program is worth a thousand words", most of this month's column is the program itself, sprinkled liberally with comments. Please take the time to study the program. But first, a few introductory words.

### Some C Basics for the Macintosh

Many data structures that you'll be working with are stored in the "heap", dynamic memory. Most of these structures are accessed by a "handle" rather than a pointer. A handle is a pointer to a pointer. Why this extra level of indirection?

The heap is used for almost everything. Code goes there. Resources go there (code is really a resource). Data structures allocated at run time go there. Nothing is preallocated. The memory manager is rather dumb about allocating and deallocating blocks of space in the heap.

As a result, the heap can (and does) get chopped up into small pieces. When someone makes a request for a block of space that exceeds the size of the largest free block, the memory manager *compacts* the heap. It moves blocks of data around in an effort to collect enough contiguous space to satisfy the request. Blocks that are eligible to be moved are called *relocatable*. Most Mac data structures fall into this class. You can control whether or not your own data structures are relocatable. The tradeoff should be obvious.

Relocatable data structures are accessed via handles so that your references remain valid if the structure is moved. The handle points to a *master pointer* which in turn points to the relocatable structure. If the structure is moved, the memory manager updates the master pointer and your handle reference remains valid. Obviously, the area containing the master pointers is *not* relocatable.

Accessing data structures defined by struct statements via handles isn't too difficult. The example is shown in figure 1 above.

The expression assigns the value contained in the relocatable structure via its handle to "intval". The only safe way to access relocatable structures is to use the double-indirect access shown above at all times. Do not attempt to

```
struct foo
{
 int e1;
 char *e2;
};

struct foo **foo_handle;

(later)

intval = (*foo_handle)->e1;
```

fig. 1

cache the pointer (\*foo\_handle) at any time. To do so is to defeat the purpose of handle access.

### Template Application

Now to the template application. It is written for the Consulair Mac C system and Toolkit. Changes to the details of the program will probably be necessary for other languages since there is little consistency in the toolbox interface.

| JOB FILE |            |      |      |
|----------|------------|------|------|
| C        | TMain.C    | Exec | Edit |
| C        | TMenu.C    | Exec | Edit |
| Link     | Templ.LINK | Exec | Edit |

fig. 2

### Job Control File

The job control file shown above in figure 2 is used by the EXEC to execute stored keyboard commands. Each line is executed by calling the application, passing the file name, and

then returning to the EXEC for the next line if successful, or invoking the editor if there is an error. The four entries of each line are the application, the file name, followed by the applications to transfer to if the operation succeeds or fails. The file name for the job control file has the ".file" extension, and is executed by the EXEC program. In this example, we will compile two C programs, TMAIN and TMENU, following which we will attempt to link them by executing the linker file TEMPL.LINK. The job control file is similar to old Apple II exec files.

```
!START
/OUTPUT Template
MacCLib
TMain
TMenu
$
```

**fig. 3**  
**Linker File**

```

* TEMPL.H *

```

```
* Common stuff for template program
* (C) 1984, MacTech by Robert B. Denny
```

```
* NOTE: Dependent on the menu
* structure defined in the resources.
*/
```

```
// Toolbox traps must start with '#' now.
#Options -N
```

```
//
// Include Macintosh data structure
// definitions. These files are included
// with the Mac C Toolkit, or from the
// Stanford University Macintosh C
// Programming system (SuMacC).
//
```

```
#include <MacDefs.H>
#include <QuickDraw.H>
#include <Control.H>
#include <Events.H>
#include <Menu.H>
#include <TextEdit.H>
```

```
#define MAX_WINDOWS 8
#define MAX_MENUS 8
```

```
#define TRUE 1
#define FALSE 0
#define NULL 0
```

```
/*
* Each window requires a window
* control block to define it's structure.
```

```
* Note which things are handles and
* which are pointers.
```

```
* wcb = Window Control Block
*/
```

```
struct wcb
{
 WindowPtr wp; // Pointer to window rec in heap
 Rect drag_rect; // Dragging limits
 Rect grow_rect; // Window size limits
 TEHandle te_handle; // Handle to textEdit rec in heap
 Point te_origin; // Text origin
 ControlHandle vs_handle; // Vert scroller's handle
 ControlHandle hs_handle; // Horiz scroller's handle
};
```

```
//
// Suppress extern declarations in module
// containing the real declarations.
```

```
//
#ifndef GLOB_DATA
extern struct wcb dw[MAX_WINDOWS];
extern struct wcb *dwp; // --> Current doc-window
extern unsigned n_windows; // No. of open windows
extern EventRecord Event; // Our event record (duh)
extern short EvType; // Event result type
extern WindowPtr EvWindow; // Event window
```

```
/*
* Array of menu handles
*/
```

```
extern MenuHandle menus[]; // Our menu contexts
#define APPLE 1
#define FILE 2
#define EDIT 3
#define OPTIONS 4
#endif
```

```
/*
* Resource ID's of menus in rsrc file.
*/
```

```
#define APPLE_ID 1
#define FILE_ID 256
#define EDIT_ID 257
#define OPTIONS_ID 258 // Add more menu resource ID's
here
```

```
/*
* File menu item numbers
*/
```

```
#define FM_EXIT // This should reflect file menu rsrc
```

```
/*
* Edit menu item numbers
```

```
* Note that SystemEdit expects the Edit
* menu items as follows:
```

```
*
* 0: undo (** YES, zero-based **)
* 1: -----
* 2: cut
* 3: copy
```

```

* 4: paste
*
* The rest of the items may be chosen
* at will by the application.
*/

```

```

#define EM_UNDO 1
#define EM_CUT 3
#define EM_COPY 4
#define EM_PASTE 5
#define EM_CLEAR 6

```

```

/*
 * Options menu assignments
 */

```

```

#define MM_EDIT_WINDOW 1
#define MM_SCRAWL_WINDOW 2
#define MM_MUSIC 4

```

```

/*
 * Other resource ID's
 */

```

```

#define ABOUT_ID 256
#define WINDOW_ID 256

```

```

/*
 * Macro to hack the Point struct,
 * composed of 2 short (16-bit) ints
 * into a long, which can be passed by
 * value to toolbox routines.
 * Takes address of a Point and turns it
 * into a scalar long. NOTE
 * this generates NO runtime code.
 */
#define PtLong(ptr) *((long *)ptr)

```

### MAIN PROGRAM A Template Application in C

The template program for the C Workshop has hooks for multiple dynamically allocated windows (to be added in future columns), supports desk accessories, has a menu bar with 4 menus and an "about" dialog in the Apple menu. The program will eventually support multiple dynamically allocated text edit and drawing windows. The text edit windows will have scroll bar controls and resizing ("grow") capability. Also, an interface to the sound system will be added. The components of this program are:

|            |                                                   |
|------------|---------------------------------------------------|
| TMMAIN.C   | This module, the main program & global data.      |
| TMENU.C    | Menu handling functions, fig. 6.                  |
| TEMPL.H    | Common "include" file of definitions, fig. 4.     |
| TEMPL.R    | RMAKER source for program resources, fig. 5.      |
| TEMPL.LINK | Linker command file, fig. 3.                      |
| TEMPL.JOB  | Exec command procedure to build template, fig. 2. |

Additional modules will be added in future C Workshop columns. This program was written for the MAC C system

by Consulair Corp. Changes will be needed for other C systems. Also, prerelease versions of MAC C and RMAKER may not work with this program. [Note: one or two obscure Mac C statements may have been changed in the current Mac C so that a change may be needed to fully compile this program using present Mac C systems. This program was written with an early version of Mac C. Any offending lines should be obvious at compile time. -Ed.]

```

/*
 * TMMAIN.C - Template application
 * (c) 1984 MacTech by Robert B. Denny
 */

```

#### GLOBALS DEFINITIONS

```

#define GLOB_DATA
#include <Templ.H>

```

```

/*
 Define window control block for window management. See
 Templ.H file.
 */

```

```

struct wcb dw[MAX_WINDOWS]; // our windows
struct wcb *dwp = NULL; // --> Current window
unsigned n_windows = 0; // Number of open windows
EventRecord Event; // Our event record
short EvType; // Event result type
WindowPtr EvWindow; // Event window
MenuHandle menus[MAX_MENUS]; // Our menus

```

#### MAIN PROGRAM

```

* Link with MacCLib, which does
* Quickdraw and Window initialization.

```

```

/*
main()
{
 register WindowPtr wp; // Try to save space/time
 int i;
 #InitFonts(); // Not done in MacCLib ...
 #InitDialogs(0); // No restart function
 #OpenResFile("011TemplRsrc");// Open resfile (Pstring)
 setup_menu(); // Set up our menus
 #TEInit(); // Initialize TextEdit
 #FlushEvents(-1,0); // Flush all events
 #InitCursor(); // Make arrow cursor
 for(i=0; i<MAX_WINDOWS; i++) // Mark all WCB's free
 dw[i].wp = NULL;
 dwp = NULL; // No windows
 n_windows = 0;

 // --- EVENT LOOP ---
 // This is the "outer loop" of a typical
 // application. All event types
 // are dequeued, most are processed.
 // Those not processed are ignored.
 //
 while(TRUE)
 // Loop here forever
 {
 #SystemTask(); // Handle desk accessories

```

```

 // dwp always points to the WCB
 //for the active (front) window.
 // If dwp = NULL, than it's not
 //our" window.
 // (Is this possible??)

dwp = (struct wcb *)find_wcb(#FrontWindow());
if(dwp != NULL && dwp->te_handle != 0) // If active wind & text
#TEIdle(dwp->te_handle); // Flash the insertion mark
if(!#GetNextEvent(-1, &Event)) // Process only our events
continue;
 switch(Event.what) // Dispatch on event type
 {
 case mouseDown: // Mouse click
 do_mouse(); // Dispatch on mouse location
 break;
 case keyDown: // Keypress
 case autoKey: // Repeat-key generator
 do_key();
 break;
 case updateEvt: // window needs refresh
EvWindow = (WindowPtr)Event.message; // Which window?
 upd_wind(EvWindow); // Do the update
 break;
 case activateEvt: // Activate/deactivate
EvWindow = (WindowPtr)Event.message; // Which window?
 if(Event.modifiers & 1) // If Activate
 act_wind(EvWindow); // Activate window
 else // If Deactivate
 deact_wind(EvWindow); // Deactivate window

// Additional cases for other event types
 default: // Do nothing for other events
 }

 }

//
// --- END OF EVENT LOOP ---
//
}

/*
 * * LOCAL FUNCTIONS *
 * Handle mouse-down events
 */
do_mouse()
{
 short window_part; // FindWindow result code
 //
 // Following fills in EvWindow with
 // --> window in which the click
 // occurred (or NULL if none).
 //
 window_part = #FindWindow(PtLong(&Event.where),
&EvWindow);

 switch(window_part) // Dispatch on where clicked
 {
 case inMenuBar:
 do_menu(#MenuSelect(PtLong(&Event.where)));
 break;
 case inSysWindow:

```

```

 #SystemClick(&Event, EvWindow);
 break;
 case inContent: // Click in content of a window
 #SelectWindow(EvWindow); // Activate window
 #SetPort(EvWindow); // Hook QuickDraw up
 break;
 case inDrag:
 #SelectWindow(EvWindow); // Activate
 #SetPort(EvWindow); // Hook QuickDraw up
 break;
 case inGoAway: // Go-away handling
 break;
 case inGrow: // Grow handling
 break;

// Add dispatches for other area types
 default:
 }

}

/*
 * - Handle keypress and command keys
 * Handles edit command keys
 */
do_key()
{
 unsigned short ch;
 ch = (unsigned short)Event.message; // Isolate key
codes
 if(Event.modifiers & cmdKey) // If command character
 do_menu(#MenuKey(ch)); // Try menu shortcut cmd
 }

// Stub functions
upd_wind(wp)
WindowPtr wp;
{
 return(0);
}

act_wind(wp)
WindowPtr wp;
{
 return(0);
}

deact_wind(wp)
WindowPtr wp;
{
 return(0);
}

find_wcb(wp) // Never one of our windows
WindowPtr wp;
{
 return(0);
}

```

**NOTE: MENU SUB-PROGRAM BEGINS NEXT  
PAGE**

## MENU SUB-PROGRAM IN C

```

/*
 * TMenu.C - Handle menu selections for
 * template program
 *
 * (c) 1984, MacTutor by Bob Denny
 */
#include <Templ.H>
/*
 * SETUP_MENU() - Initialize menu system
 *
 * Fills in an array of menu handles. Not
 * used in this version of the program.
 */
setup_menu()
{
 #InitMenus();
 #InsertMenu(menus[APPLE] =
(MenuHandle)#GetMenu(APPLE_ID), 0);
 #AddResMenu(menus[APPLE], 'DRVr');
 #InsertMenu(menus[FILE] =
(MenuHandle)#GetMenu(FILE_ID), 0);
 #InsertMenu(menus[EDIT] =
(MenuHandle)#GetMenu(EDIT_ID), 0);
 #InsertMenu(menus[OPTIONS] =
(MenuHandle)#GetMenu(OPTIONS_ID), 0);
 #DrawMenuBar();
}

/*
 * DO_MENU() - Handle menu selection
 *
 * Input:
 * Result longword from MenuSelect
 * or MenuKey
 */
do_menu(result)
unsigned result;
{
 unsigned short menu_id; // Resource ID of selected menu
 unsigned short item_no; // Item number selected
 char item_name[64]; // Item name (for desk acc.)
 Ptr dp; // Dialog pointer for "about ..."
 unsigned short item_hit; // Dialog item that was hit
 if(result == 0) // Just for safety with MenuKey
 return; // Ignore zero results
 menu_id = (short)#HiWord(result); // Use toolbox
 item_no = (short)#LoWord(result);

 switch(menu_id)
 {
 case APPLE_ID: // "Apple" menu
 if(item_no > 1)

 // If desk accessory
 {
 #GetItem(menus[APPLE], item_no, item_name);
 #OpenDeskAcc(item_name);
 }
 else // Our About ... dialog
 {
 dp = #GetNewDialog(ABOUT_ID, 0, -1);
 // Bring in dialog to front
 #SetPort(dp); // Hook QuickDraw up to dialog
 #ModalDialog(0, &item_hit); // dialog item hit
 #DisposeDialog(dp); // Close & free heap
 }
 break;

 case FILE_ID: // File menu
 if(item_no == FM_EXIT)
 #ExitToShell(); // Ugly exit
 break;

 case EDIT_ID:
 //
 // First we must hand off any
 // desk-accessory edit commands.
 // Note the comments in Demo.H
 // regarding implicit assumptions
 // in numbering items in Edit menu.
 //
 if(#SystemEdit(item_no - 1)) // Relies on item no.!!
 break;

 //
 // Do the requested edit function only
 // if this is an editing window and there
 // is a window open. For now, we have
 // no windows, so we just return.
 //
 break;

 case OPTIONS_ID: // Options menu.
 switch(item_no)
 {
 case MM_MUSIC:
 play_tune();
 break;
 case MM_EDIT_WINDOW:
 edit_window();
 break;
 case MM_SCRAWL_WINDOW:
 scrawl_window();
 break;

 default:
 break;

 default:
 #HiliteMenu(0); // Turn off highlighted title
 }

 // Stub functions
 play_tune() // Make crude music -- demo driver
 {
 return(0);
 }
 edit_window() // Create a new editing window
 {
 return(0);
 }
 scrawl_window() // Create a new "scrawling" window
 }
}

```

```
{
return(0);
}
```

## Resource File

This is the RMAKER source for the Mac C template application. The dialog box will not come out right with the version of the RMAKER that I have. The \0D entries in the staticText item are supposed to be "hard" carriage returns in the text. Instead, they come out as "D". To fix this, use the AlertDialog Editor on the file after building it to put the 2 carriage returns in and remove the "D"s. Perhaps RMAKER (or it's successor) will work someday...

The menu items for things that aren't in the template (yet) have been dimmed. As things are filled in, you'll have to edit this file to un-dim & thus activate the corresponding menu items. Note that the Edit menu is enabled to support SystemEdit editing in desk accessories.

```
* Templ.R -- Resources for the
* template program
*
* (C) 1984, MacTech by R. B. Denny
```

```
TemplRsrc
RSRCXXXX
```

```
TYPE WIND
,256
Please Wait ...
50 50 250 250
Visible GoAway
0
0
```

```
TYPE MENU
,1
\14
About Template ...
(-
```

```
,256
File
Quit
```

```
,257
Edit
(Can't Undo
(-
Cut/X
Copy/C
Paste/V
Clear/D
```

```
,258
Options
(New Edit Window
(New Scrawl Window
(-
```

(Play a tune

```
TYPE DLOG
,256
```

```
54 145 203 376
Visible NoGoAway
1
0
256
```

```
TYPE DITL
,256
2
```

```
button
116 58 142 174
RESUME DEMO
```

```
staticText Disabled
9 9 105 228
```

```
This program was written as an example of native Macintosh
++
application programming in the C language, using Mac C \0D
\0D ++
```

```
Bob Denny October, 1984
```





# C Workshop

## Planning for Desk Accessories



Bob Denny  
Editorial Board  
MacTutor Vol. 1 No. 3

Last month's C Workshop presented an application "template", a program shell which can be used as the basis of many kinds of applications. The template can be used with most C systems since there are no calls to library functions. The only external services used are native Mac toolbox traps.

This month, we'll look at some of the basics in constructing applications, including the tricky area of servicing desk accessories properly. If you have access to last month's C Workshop, you can refer to the template application's code for examples. It's not necessary, though.

### Common Requirements

Most applications share several common requirements. These are:

- 1) Overall control is via the menu bar and pull-down menus.
- 2) Detail control and input uses both the mouse and the keyboard.
- 3) Multiple instances of multiple types of windows serve as user interfaces.
- 4) All desk accessories must be usable during execution of the application, within the limits of available memory.

These requirements, combined with the Macintosh operating system, imply the overall application setup and control structure described below.

Applications should begin by initializing all of the system services that it intends to use. At a minimum, all applications should call `InitGraf()`, `InitWindows()`, `InitFonts()`, `InitMenus()`, `InitDialogs()` and `TEInit()`. Some desk accessories use dialogs, which in turn use `TextEdit`.

### Menu Setup

Any application that supports desk accessories *must* support a menu bar with at least the "Apple", "File" and "Edit" menus. These menus must be present as required by the Macintosh User Interface Guidelines, and desk accessories assume that they are present. Moreover, the "File" and "Edit" menu options must also conform to the User Interface Guidelines. Desk accessories make assumptions as to the order and meaning of options in these menus.

The "Apple" menu must be set up to contain the desk accessories for selection and opening. Your application must

do this explicitly by first creating the menu then adding the desk accessories via the `AddResMenu()` service. For example:

```
MenuHandle mh;
...
InsertMenu(mh = GetMenu(1), 0);
AddResMenu(mh, 'DRVr');
...
DrawMenuBar();
```

In the example, `GetMenu()` loads the menu resource whose ID = 1 and returns a handle to it, which gets stored in `mh`. `InsertMenu()` then adds the menu to the menu bar. Next, `AddResMenu()` appends the names of all resources of type 'DRVr' to the menu. Desk accessories are "drivers" to the current Mac operating system, hence have the resource type of 'DRVr'. Real drivers have names beginning with '.' or '%' and will not get added to the menu.

The "File" menu is not used (yet) by any of the standard desk accessories. It is up for grabs, though, and should follow

the User Interface Guidelines. That is, the items should follow the order:

```
New
Open...
Close
Save
Save as...

Page Setup
Print...

Quit
```

Desk accessories are free to assume that the ordering (and therefore numbering) of menu items in the "File" menu conforms to the above.

The "Edit" menu is used by the Note Pad and other desk accessories. Again, the layout of the menu must follow the user interface guidelines, namely:

```
Undo

Copy
Cut
Paste

Show Clipboard
```

To reiterate, the leftmost three menus must conform to the Macintosh User Interface Guidelines stated in Inside Macintosh if the application is to support desk accessories.

## The Event Loop

The single most important characteristic of a Macintosh application is that it is *event-driven*. Most Mac applications have an "event loop" as their outermost control structure. Following initialization, the application does something like:

```
while(TRUE)
{
 SystemTask();
 if(!GetNextEvent(-1, &event))
 continue;
 switch(event.what)
 {
 ... cases for each event type
 ... our application handles

 default: /* Junk other events */
 }
}
```

It is vitally important to understand the event loop. Most trips around the loop circle back at the `continue` statement that gets executed if `GetNextEvent()` returns `FALSE`, meaning that there is no event for us to handle. In future versions of the Macintosh operating system which support multitasking, `GetNextEvent()` will probably return control to the scheduler, giving other tasks CPU time until an event occurs for the caller.

In any case, when an event for us does occur, `GetNextEvent()` returns `TRUE` and fills in our *event record* with information describing the nature of the event. In C, the event record can be defined as follows:

```
struct EventRecord
{
 short what;
 long message;
 long when;
 Point where;
 short modifiers;
};
```

where the type `Point` is some mapping of the QuickDraw "point", an ordered pair of 16-bit coordinates.

Control passes to the `switch()` statement, which dispatches to the event handling function appropriate for the event type. For example, an event type of `mouseDown` would dispatch to the function that handles mouse clicks.

In the example shown, the value for the first parameter to `GetNextEvent()` is `-1`. This parameter is the *event mask*, and in this case, it is set to "get" all types of events. The `switch()` statement should have case sections for each event type that

the application explicitly handles. Other types of events use the default case, which does nothing.

If the event mask parameter specifies a subset of event types, those not specified will remain on the event queue for possible later processing. This may be needed for applications which use a certain event type to trigger processing of earlier events of other types not normally processed. This is tricky, though, and can be the source of some obscure bugs. Also, unprocessed events on the queue take up valuable memory space.

If that isn't enough, there's a system-wide event mask that controls what event types are recognized and queued. Your application can disable the queueing of certain event types by calling the `SetEventMask()` service. If your application doesn't process certain event types, you should disable them in the system event mask. This prevents wasting valuable event queue memory space with unprocessed event records.

The bottom line is... call `GetNextEvent()` with an event mask of `everyEvent (-1)` to dequeue all event types, then let the default case dispose of the events you aren't interested in.

Just prior to calling `GetNextEvent()`, notice the call to `SystemTask()`. This Mac system service causes control to pass to each open desk accessory, in turn, then back to the caller. Normally, one call at the beginning of the event loop is sufficient to give desk accessories the time they need. If you have any places in your application where you "spin your wheels", consider calling `SystemTask()` to help waste some time.

In a multi-tasking version of the Mac, `SystemTask()` will probably do nothing. Rather, control will be given to the task only when there is a live event for that task. Timer services will be provided for passing control back to the scheduler (and other tasks) when a synchronous delay is needed in the current task.

The event loop is the fundamental control structure of a Macintosh application. If you haven't studied **The Event Manager: A Programmer's Guide**, a chapter in Inside Macintosh, you should take the time now to do so.

## Mouse-Down Events

Any application which supports desk accessories *must* handle mouse-down events. A mouse click in a desk accessory window gets passed as an event to the application. This quirk in the Mac system architecture requires the application to determine whether the click was in a desk accessory window or in one of its own windows. I should mention that this approach to click handling minimizes the uncontrollable overhead in the operating system for applications which need every available CPU cycle and which do not support desk accessories (whew!).

So, if you want to support desk accessories, your application must detect and process mouse-down events. If you detect a mouse click, first call the Window Manager function `FindWindow()` to find out where the cursor was when the mouse button was clicked.

If FindWindow() returns the predefined constant inSysWindow, it means that the cursor was in a desk accessory window when the mouse was clicked. If this is the case, call the Desk Manager function SystemClick(). This passes control back to the operating system service which handles desk accessory windows.

Other returns from FindWindow() indicate mouse clicks in windows (including what "part" of the window) or the menu bar, or out in "no man's land", the desktop background. We'll discuss window handling in a future C Workshop. If the click is in the menu bar, then our application must dispatch to a menu handler, another necessity if we are to handle desk accessories.

### Menu Selections

If the mouse click is "in the Menu Bar", then our application must first determine the selected menu number and the item number in that menu, then dispatch accordingly. First, call the MenuSelect() service to get a longword containing both the menu number and the item number in that menu, then use HiWord() and LoWord() to split them up:

```
unsigned short menu_id, item_no;
unsigned long result;
unsigned short HiWord(), LoWord();
unsigned long MenuSelect();
...
result = MenuSelect(&event.where);
menu_id = HiWord(result);
item_no = LoWord(result);
```

Then use nested switch() statements to dispatch based on the menu and item selected.

If the "Apple" menu has the desk accessories, your application must activate a selected accessory. If the menu selected is the "Apple" menu, and you have determined that the selection was indeed an accessory, then you simply call OpenDeskAcc() with the name of the menu item (the desk accessory name). The latter can be obtained by calling GetItem() with the menu item number.

The "Edit" menu must also be handled specially if desk accessories such as the Note Pad are to be supported properly. Before trying to dispatch to your own edit-handling functions, you must call the SystemEdit() service. If it returns FALSE, then dispatch to your own function. If it returns TRUE, however, then the menu selection was made while an editing desk accessory was open. The desk accessory handled (used) the request; you must ignore it.

If you haven't yet studied the The Desk Manager: A Programmer's Guide, a section of Inside Macintosh, then you should do so now. It's required reading for programmers who are developing applications which must support desk accessories.

### Final Words: Resources Versus Static Data

I'd like to finish up this month's C Workshop with some thoughts on the use of resources in C programs. Pascal has no facilities for statically initializing data structures. Therefore, Pascal program- mers have no choice but to load static data via the resource mechanism if they are to avoid ugly space-consuming "initialization segments". You should carefully consider the tradeoffs before deciding whether to make a particular data structure static in your program or to place it in a resource.

Any data that is designed to be tailored after building the application belongs in a resource. Same with large data structures that are used infrequently and can be "purged" from the heap. They too should go in a resource.

It's my feeling that anything else belongs in your application's static data area if you can afford the room. Why? The use of resources can markedly slow down your application. On 128K systems, heap space is fragmented by frequent resource manager activity, causing garbage collection cycles and purges to disk. Reading in of resources to initialize static data is far slower than having them present in your application's address space when it is loaded.

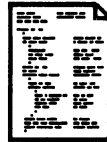
Last but not least, it is difficult to maintain programs which contain constants that depend on the contents of resources. For example, the numbering of menu items must track across the C code and the RMaker source as well. As a C programmer, you have a choice.



# C Workshop

## Mac's Window Technology

C



Bob Denny  
Editorial Board  
MacTutor Vol. 1 No. 4

### Layers and Interfaces

One of the most important things to come out of the computer world in the Seventies is the concept of **layered architecture**. Layering provides us with a clean way to deal with the complexities of modern operating system and communications architectures. One of the most visible instances of layering is the ISO Open Systems Architecture which is widely used and quoted in standards for networks.

I consider layering to be a survival skill for software developers. Take time to consider the Macintosh from bottom to top. At the bottom, we have solid-state physics which attempts to explain the behavior of what might be particles or waves. At the top (user's view) we have dragging, clicking, icons, windows, etc. Both the physics and the user concepts are abstract ideas. How many layers can you conceive of in a Macintosh?

Let's formalize this a bit. A **layer** is a structured set of related things that together provide a set of tools or services for implementing other things. An **interface** is a set of means for accessing the services provided by a layer. Finally a given layer might make use of the services provided by a "lower" layer in implementing it's own services. It is this latter notion that makes layers have ranking.

How does all of this help us deal with complex systems? By restricting our thinking to the layer we are working on, we avoid cluttering our mind with unnecessary details. This kind of thinking comes naturally, but we can reinforce it by being aware of its importance.

Most of the developers I have talked with share the feeling that the Mac is a difficult system to learn. Inside Macintosh is a wealth of information, but it has little overall structure. There is a section called Inside Macintosh: A Road Map which does discuss the overall architecture, but this thread is not carried elsewhere, and one tends to "dive in" to a particular area of interest.

Here is my personal view of the Macintosh operating system as a set of layers. Below these layers are the low-level stuff I seldom deal with and above these layers are the programs I write. The boxes show the sections of the Mac system that I frequently encounter when developing user-oriented applications.

### Data-Driven Software

Another key concept needed for Macintosh programming is that of data or table-driven software. This is another common technique that grew out of the Seventies, when system complexity took a quantum jump.

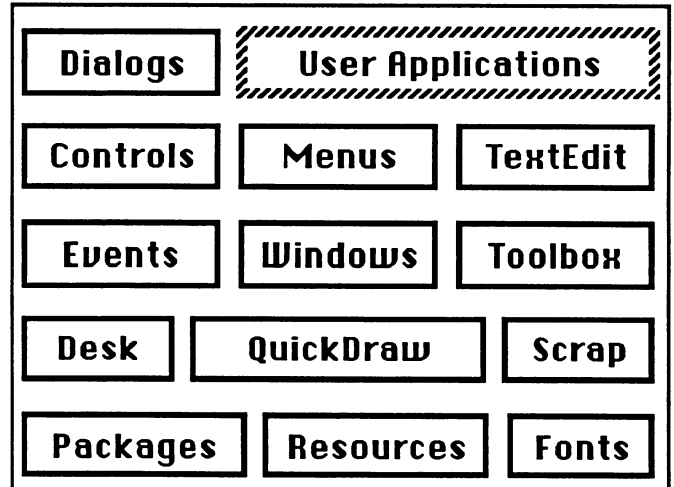


Fig. 1 The Mac Environment

A data-driven routine consists of a combination of code and data, split up so that variations are possible with no changes to the code. This technique is usually employed where there are a variety of related objects that need some sort of service or management that can be done with a common "engine" in code.

This is sort of a gray area; you might say that all routines are sensitive to data, or they wouldn't be useful. True, but a routine that stores no data of its own is special. It is usually called with a reference to a data structure which contains the state and attributes of the particular object that the routine is to work on. The data structure could even contain a reference to one or more "action routines" which are unique to the object in question. Examining the contents of such data structures can often provide great insight into the subtleties of the routine or service.

Most of the "managers" in the Macintosh interface use this technique. The data structures are referred to as "records" from the Pascal terminology. In C, they can be implemented as struct's.

### The Window Manager

From the user's viewpoint, a window is an object on the desktop that provides some sort of user interface. There is a whole set of guidelines describing how windows should look and behave. Window appearance and behavior is the essence of the Macintosh.

From the developer's viewpoint, a window is a structured set of regions in a port under the control of the **Window Manager**. The Window Manager provides a layered interface to window services on the Macintosh, using QuickDraw to

draw the pictures that create the abstract notion of windows on the screen.

Besides handling the windows themselves, the Window Manager handles shuffling, dragging and re-sizing of overlapping windows, generating events for owning applications to update window contents as required. This powerful and complex layer of the Mac architecture has a surprisingly simple interface.

The Window Manager uses a data structure, called a **window record**, to keep track of each window. It contains the current attributes and state of the corresponding window. The window records are kept on a linked list whose order corresponds to the planar ordering of the window images on the screen, as shown in figure 2.

If you haven't looked over the section on the Window Manager in *Inside Macintosh*, now would be a good time to do so. In particular, study the section describing the window record and each of its fields. We'll look at some of the more important fields here, but we can't cover them all. In C, the window record can be represented as follows:

### C Struct for Window Records

```
#define boolean char
struct WindowRec
{
 grafPort port;
 short windowKind;
 boolean visible;
 boolean hilited;
 boolean goAwayFlag;
 boolean spareFlag;
 RgnHandle strucRgn;
 RgnHandle contRgn;
 RgnHandle updateRgn;
 Handle windowDefProc;
 Handle dataHandle;
 Handle titleHandle;
 short titleWidth;
 Handle controllList;
 struct WindowRec *nextWindow;
 PicHandle windowPic;
 long refCon;
};
```

Note that the first thing in the window record is a **GrafPort** structure. This is the data structure used by QuickDraw for manipulating "ports" in general. Of course, a window is a port, so it follows that the window's data would include (nested within) a GrafPort.

Before we dig any deeper into the window record, let's look at how it relates to other important data structures. Pointers and handles are represented alike in the interest of simplicity in Fig. 3.

### Ports, Regions and Windows

Looking at figure 3, observe the linkage of the region structs. The visRgn and clipRgn are hooked to the GrafPort,

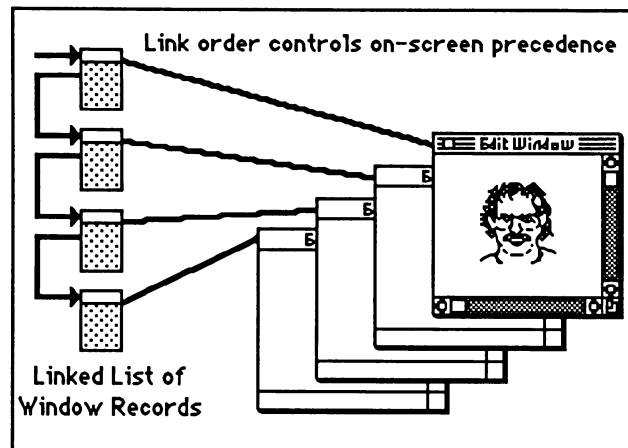


**Fig. 2 Screen Precedence**

while the strucRgn, contRgn and updateRgn are hooked to the WindowRecord. The latter three regions are used only in the context of a window. If you are unclear about the role of regions, refer to Chris Derossi's *Pascal Procedures* in *MacTutor*, Vol.1, No.3, February, 1985.

Note also that the ControlRecord for each control that is associated with the window is hooked to a linked list, and has a "back-link" pointing back to the owning window's WindowRecord. This implies that the Window Manager uses information about the controls, and that the Control Manager uses information about the owning window and its port. For example, the Window Manager function drawControls() traces down the linked list of ControlRecords, drawing each control in turn.

The various "boolean" fields in the WindowRecord indicate the current state of the window's visibility, highlighting (see below) and whether it has a "go-away" box. **CAUTION:** the Window Manager uses 255 as TRUE and 0 as FALSE. Typically C defines TRUE as 1. You should test these flags for "not FALSE", rather than explicitly TRUE.

The three RgnHandles point to Rgn structs which describe important parts of the window's port. We'll cover these later in the section on anatomy and next month's column on window dynamics.

The windowDefProc field is a handle that points to a vectored set of routines that give the window its look. Associated with this is the dataHandle field, which points to private data used by the DefProc. "DefProc" is short for Definition Procedure, really a set of routines that perform the following:

### Window Definition Procedure Tasks

- Draw window frame
- Return region where mouse was clicked
- Calculate structure & content regions
- Draw "grow" image of the window
- Draw a "size box" in the content region

Note that the Window Manager has no idea what the window actually looks like. The appearance is completely

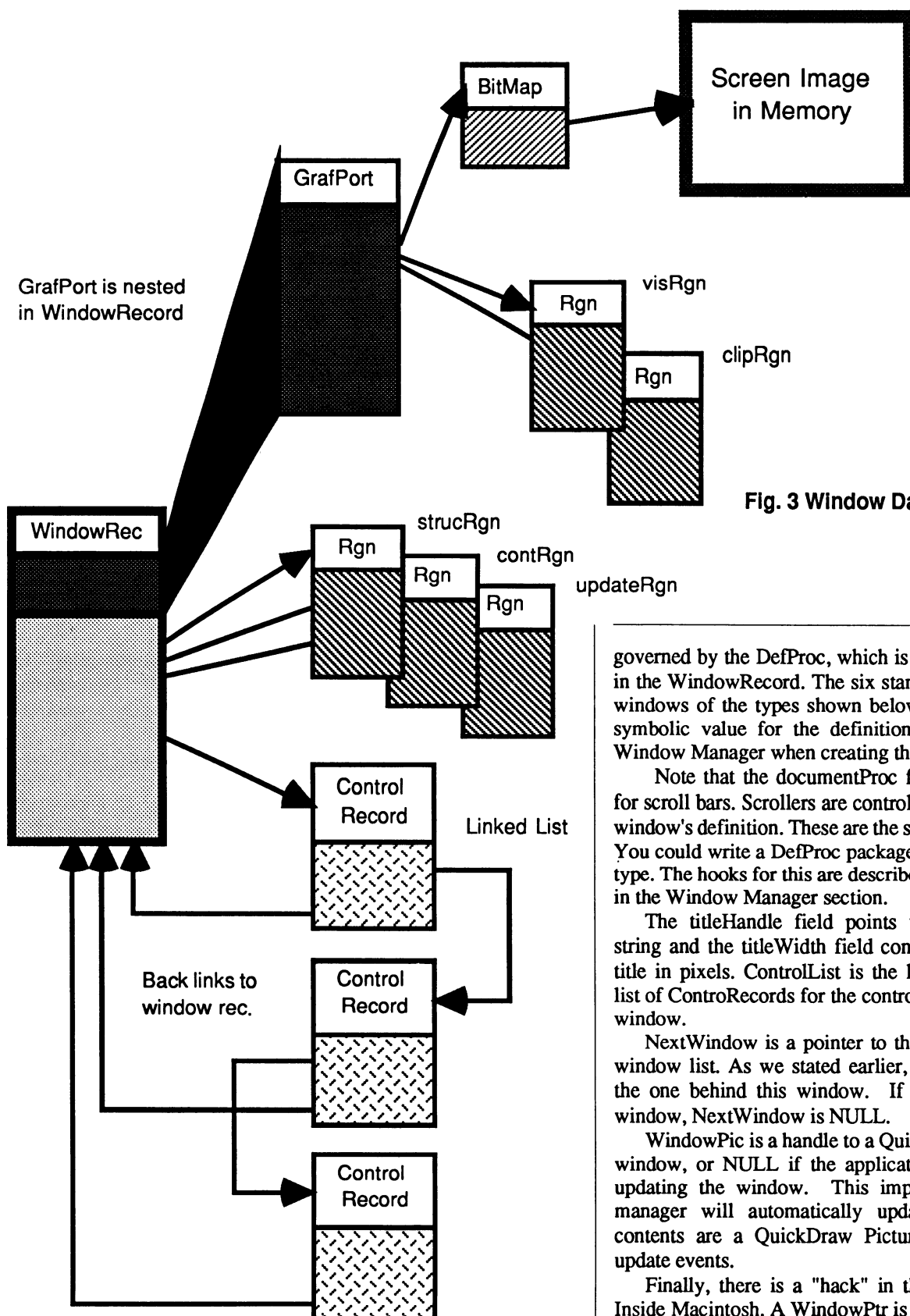


Fig. 3 Window Data Structures

governed by the DefProc, which is described by a handle in the WindowRecord. The six standard DefProcs handle windows of the types shown below in figure 5 with the symbolic value for the definition ID you pass to the Window Manager when creating the window.

Note that the documentProc flavor has the outlines for scroll bars. Scrollers are controls, not part of the basic window's definition. These are the standard window types. You could write a DefProc package for a custom window type. The hooks for this are described in *Inside Macintosh* in the Window Manager section.

The titleHandle field points to the window's title string and the titleWidth field contains the width of the title in pixels. ControlList is the listhead for the linked list of ControlRecords for the controls associated with this window.

NextWindow is a pointer to the next window on the window list. As we stated earlier, the "next" window is the one behind this window. If this is the back-most window, NextWindow is NULL.

WindowPic is a handle to a QuickDraw picture for the window, or NULL if the application is responsible for updating the window. This implies that the window manager will automatically update a window if its contents are a QuickDraw Picture without generating update events.

Finally, there is a "hack" in the definitions used in *Inside Macintosh*. A WindowPtr is defined as a pointer to a grafPort, rather than a pointer to a WindowRecord. The name for the later is WindowPeek. It seems that the most frequent use of WindowRecord information involves

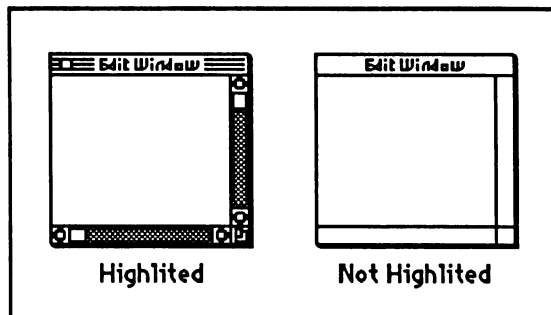


**Fig. 4 Highlighting Windows**

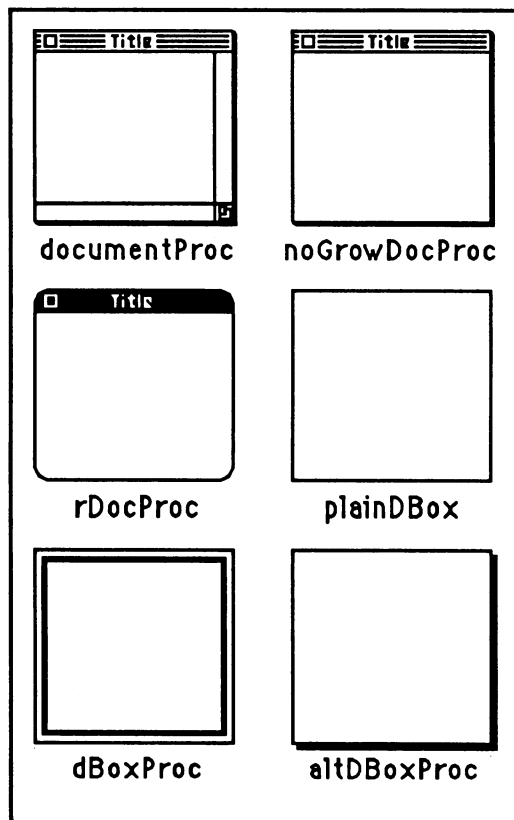


**Fig. 5 Window Types**

the data contained in the embedded grafPort. Keep this quirk in mind.

### Anatomy of a Document Window

The six standard window types are shown in figure 5. Of the six, the documentProc type is the most complex. In this section we'll look at the window as a set of regions. Viewing the window this way is essential when writing programs which manipulate them, particularly the routines that handle activation and updating of the window. A document window with scroll bars is shown in figure 6.

All windows have a structure region and a content region.

The structure region covers the entire window, frame and all. The content region is the area inside the frame. These regions change only when the window is moved or re-sized.

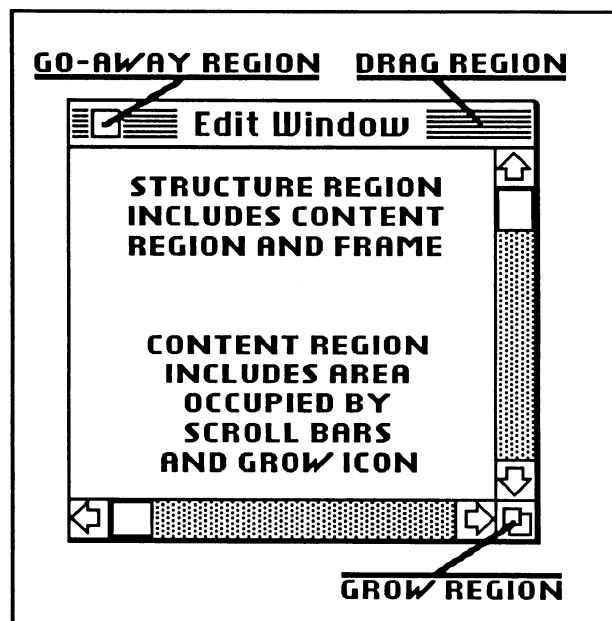


**Fig. 6 Window Construction**

The frame may include go-away and drag regions. The content region may include a grow region. These are not really regions in the formal sense, as defined by QuickDraw. The DefProc is responsible for handling these regions, not QuickDraw. The scroll bars are handled by the Control Manager, not the Window Manager.

The WindowRecord contains a handle to another region, the Update Region. This region describes the portion of the content region that needs updating, redrawing, after some change. We'll deal with this in next month's column on window dynamics.

One of the fine points not made clear in *Inside Macintosh* is the relationship between the "grow image", the grow region and the scroll bars usually present in a document window. The grow image is the line-tracing of the window that appears when you click in the grow region.

The Control Manager can build scroll bars of any dimension. The convention is that scroll bars are 16 pixels wide. This matches the width of the "scroller" areas that appear in the grow image, and the width and height of the grow icon. These areas are fixed, set by the documentProc DefProc. Here is a fat-bits view of the lower right corner of the document window shown above (see fig. 7). Note that the scroll bar edges overlap the grow icon and the window frame by one pixel. The following fragment of C code computes the boundsRects for the scrollers, given the portRect for the window. Get the portRect from the grafPort that is embedded in the WindowRecord:

```
struct rect /* Define a "rect" */
{
```

```

short top;
short left;
short bottom;
short right;
};

```

```

struct grafPort *wp; /* Window Pointer */
struct rect *pr; /* -> portRect */
struct rect vs_rect; /* Vert. scroller */
struct rect hs_rect; /* Hor. scroller */

```

```
pr = &(wp->portRect);
```

```

/*
 * Compute bounds rect for vertical scroll
 */

```

```

vs_rect.top = pr->top - 1;
vs_rect.left = pr->right - 15;
vs_rect.bottom = pr->bottom - 14;
vs_rect.right = pr->right + 1;
/*

```

```

 * Compute bounds rect for horiz. scroll
 */

```

```

hs_rect.top = pr->bottom - 15;
hs_rect.left = pr->left - 1;
hs_rect.bottom = pr->bottom + 1;
hs_rect.right = pr->right - 14;

```

### Afterword

Next month we'll deal with window dynamics, dragging, growing, updating and activating. This is an area of Mac programming that seems complex on the surface. If you have tried your hand at window-oriented programs, you probably had many surprises and mysteries and you may have developed a few OPT's (old programmer's tales). One of my most memorable surprises was discovering what happens if you omit the beginUpdate() and endUpdate() calls when handling an update event.

There is an underlying consistency to window dynamics. It's all there in *Inside Macintosh*, but in pieces. Understanding window dynamics is essential if you are to get beyond trivial programs without unnecessary screen drawing or ugly hacks. See you then.

**SCROLL BAR EDGES OVERLAP  
WINDOW EDGES BY 1 PIXEL**

**SCROLL BARS ARE  
16 PIXELS WIDE**

**1-PIXEL OVERLAP**

**1-PIXEL DROP SHADOW**

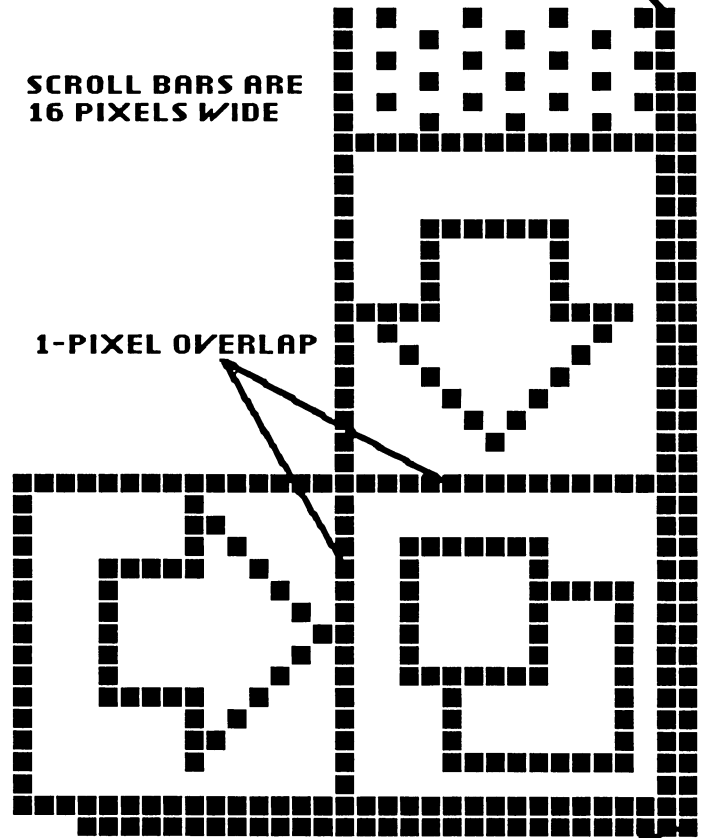


**Fig. 7 Grow Region Expanded**





# C Workshop

## Window Dynamics



Bob Denny  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 5

### Windows In Action

There is an initial "hill" that every Macintosh developer must climb before doing anything useful. Most applications require use of windows for presentation and control. Unfortunately for the developer, this fundamental requirement makes it necessary to understand a large part of the Mac's internals. This information is presented piecemeal in Inside Macintosh (IM).

Last month, we covered the "static" aspects of windows, including data structures and the functions of the Window Manager. Much of this information can be gleaned from IM relatively easily.

The dynamic aspects of windows, however, are far more subtle. Most of what we learn about window dynamics comes from our experience implementing window-based applications. Take some time to observe how various Mac applications handle things like activating overlapped windows and dragging or resizing of windows. Some applications take an "overkill" approach, erasing the entire window and redrawing everything. Others behave in a more refined fashion, updating only what's necessary.

The goal of this month's column is to provide you with some insights into window dynamics and how to apply that knowledge to avoid over-updating and ghost images.

In particular, we will cover creating, updating, dragging, resizing, activating and deactivating of windows. The details of scroll-bars, textEdit and other window "content" items will be covered in next month's column.

### Regions Again

Before we go any further, let's quickly review the various regions involved in window construction and dynamics.

Fig. 1 shows the various component regions of a common "edit" window. The terminology used here and in Inside Macintosh is confusing. The go-away, drag and grow regions aren't really "regions" in the QuickDraw sense. They are areas within the window that are handled specially by the Window Manager. For the purposes of compatibility with IM, however, we will use the same terminology.

The structure and content regions are true regions. The structure region encloses the entire window, frame and all. The frame consists of the entire title bar and the border (even the drop-shadow, if any). The content region is that area inside the frame. Note that the scroll bars and size box are located *within* the content region. These regions change only when the window is moved or re-sized.



Fig. 1 Window Regions

### The Update Region

There is another region that we will be dealing with extensively: the update region. It's not a structural part of the window, as are the structure and content regions. Instead it is a dynamic description of the area(s) of the window which need updating at any given instant.

The update region, like the structure and content regions, is linked directly to the windowRecord. It is used by the Window Manager to control QuickDraw updates to the screen. We'll look at this in more detail later.

When windows are shuffled front and back, or moved about the desktop, it can be confusing as to who is responsible for redrawing what parts of the window. What will the window manager do for you, and what must your application do itself? Here is a simple rule to remember:

**The Window Manager handles the frame: your application must handle the contents.**

The contents include any controls (e.g., scroll bars) and the "grow icon" in the grow region, as well as whatever you have placed elsewhere in the window area.

### How a Window Is Drawn

Let's look at the process of creating a new window, neglecting the programming needed and concentrating on what actually happens. Keep in mind the rule presented above regarding who handles what.

First, the Window Manager calls the "definition procedure" (DefProc) for the window type and asks it to draw the frame. In doing this, the DefProc manipulates the desktop visRgn and clipRgn to make sure that only the visible portion of the frame will be actually drawn. The DefProc may also draw a go-away box in the title bar area. If you are unclear about DefProcs, refer to last month's [C Workshop](#).

When the DefProc returns to the Window Manager (with the frame drawn), the Window Manager posts an update event for the application that requested the window to be drawn, then returns control to the application.

It is important to understand just how this update event gets generated. The Window Manager determines which areas of the window's content region need updating. This set of areas is accumulated into the window's update region, which is now no longer empty. When the Toolbox Event Manager sees a window with a non-empty update region, it queues an update event for that window.

**Note that drawing and updating is supported for any window on the desktop. The window being drawn need not be frontmost nor the "active" window. It might not be visible at all.**

Sometime later, when the application de-queues the update event just posted, it must draw the window contents. This could be tricky, since all or part of the window could be obscured, and if it's being re-drawn, some of the image might not need updating.

The proper method of handling update events involves use of two very important Window Manager services, BeginUpdate() and EndUpdate() as follows:

```
/*
 * Handle update event
 */
do_update(wp)
WindowPtr wp; /* Update this window */
{
 BeginUpdate(wp);
 ... /* Draw everything */
 EndUpdate(wp);
}
```

BeginUpdate() calculates the intersection of the window's visRgn and the update region. The result of this intersection is "the area that is visible and needs updating". It then replaces the visRgn with this newly calculated region. Finally, it clears the update region so that the update event will not be repeated for the window. Then control is returned to the update event handler.

At this point your application may draw the entire window contents without fear. Actual drawing will be restricted to the (temporary) visRgn, which encloses only what may and should be drawn. Nothing else will be drawn.

After this has been done, call EndUpdate() to restore the original visRgn. This completes the actions needed to service an update event.

So the steps are, restrict actual drawing by hacking the visRgn, draw everything, then restore the real visRgn. This approach to window content maintenance is absolutely basic to successful Mac application development.

Don't be tempted to try analyzing the hacked visRgn calculated by BeginUpdate() and manually restrict your drawing to that region only. You may think you're saving time by not drawing invisible areas, but you're more likely to chew up at least that much time figuring out what you don't want to do!

### Deferred Drawing

Window contents must be updated whenever something happens to cause a portion of the window's image to become visible after being obscured. The Window Manager will automatically accumulate areas into a window's update region, and your application may manually add to the update region as well.

In either case, when the update event occurs, part or all of the window may need redrawing. In order to do this properly, you must have a description of the window's contents stored somewhere. Think about the implications of this.

Let's say your application requests input of a set of 10 values that you use to draw a figure. You simply read the values and draw the figure as they are read in. Now let's say you drag the window partially off-screen, release it, then drag it back on-screen. You'll get an update event for the area that was off-screen. How can you reproduce the figure? The values are gone.

You should have stored the values in an array so that they could be used by the update event handler to redraw the figure as needed. And why draw any time except after an update event? This is a trivial example of an important rule in Mac programming:

**Never "store" anything in the window's image. Always maintain the data needed to redraw the window, and draw only in response to an update event.**

If you change something in the window, change the descriptive data, then force an update event by calling InvalRect() or InvalRgn() to put the changed area into the update region. Let the update event handler do the actual drawing. This method of window maintenance is called "deferred drawing" because the changes are made to the descriptive data and the actual drawing is deferred to the time of the update event.

If you follow the above rule, it will usually simplify your program's logic, making it more reliable and less likely to leave "fossils" on the screen or blow holes in the window image.

### Activation and Deactivation

The Macintosh User Interface Guidelines specify that only one window on the screen may be "active". What does

active mean? Remember that any window on the screen can (and may) be updated at any time. But the user should be able to tell which window will react to keystrokes and/or mouse clicks. Which window is "hooked up" to the mouse and keyboard?

The active window is the one the user is "working in" at the moment. It is always the frontmost window (alerts and dialogs excepted) and has a distinctive highlighted appearance.

The Window Manager provides services to activate and deactivate windows. Also, the system delivers activation events to the application so that it can make any changes to the window contents as a result of activation or deactivation. Remember, the Window Manager handles the frame and the application handles the contents.

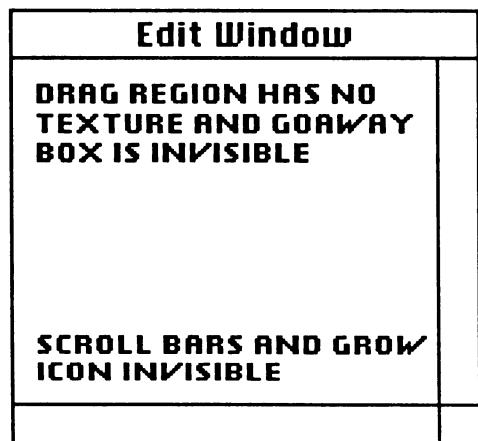


Fig. 2 Deactivate event

Fig. 2 shows a document window in its deactivated state. The title bar (drag region) has no stripes and the go-away box is not present. These items are controlled by the Window Manager. The scroll bars are hidden and the grow icon is gone. These items are controlled by the application's deactivation event handler.

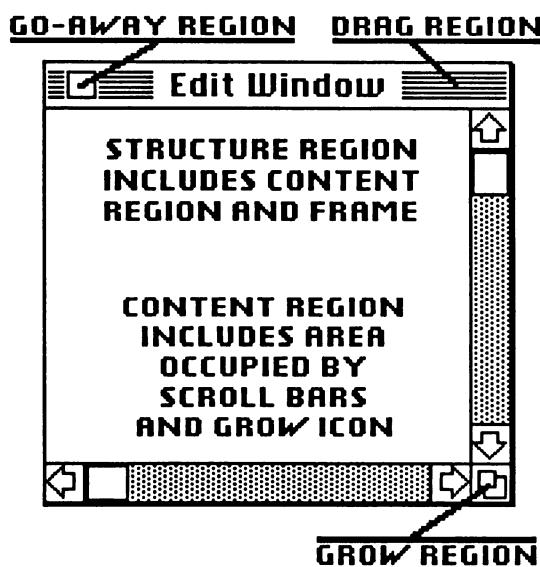


Fig. 3 Activated Document Window

Fig. 3 shows an activated document window. The title bar has stripes showing and the go-away box is visible. The Window Manager handles these items. The scroll bars and the grow icon are visible. The application's activate event handler handles the latter items.

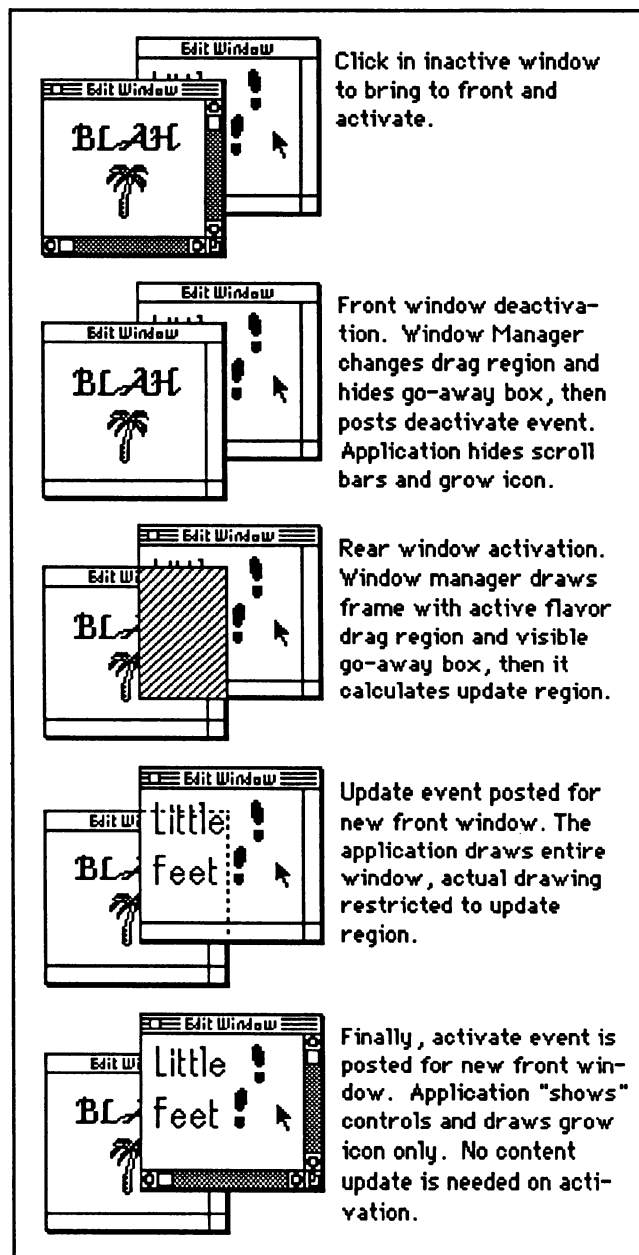


Fig. 4 Activate Event

Activation and deactivation are signalled by the same "activation" event being queued for the window. The application must decode which action to take.

What your application should do in response to update and activation events can become confusing. Servicing update and activation events is a simple matter if you keep the following rule in mind:

Drawing should be done only for update events. Activation and deactivation should cause

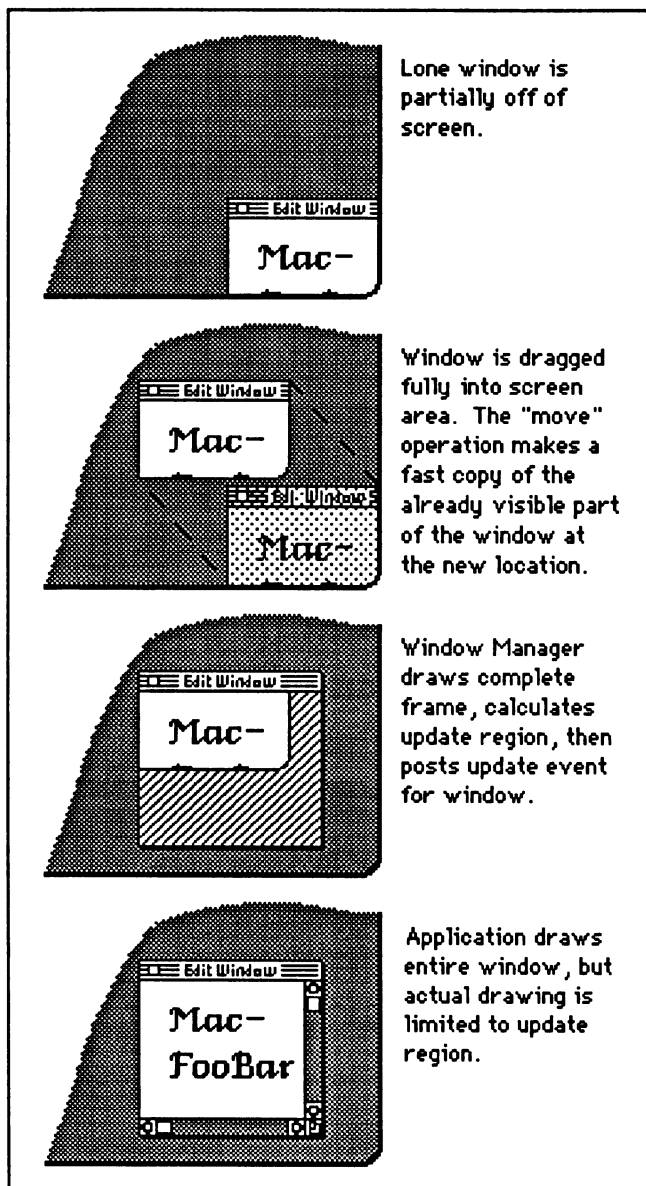


**Fig. 5 Update Event**

only changes in appearance of activation dependent items such as controls and menus.

If you follow this rule, it will simplify the logic of your program and prevent unnecessary drawing. Do actual drawing only in response to update events. Use activation events to change the appearance of items.

### Putting It All Together

Let's take the concepts we have learned and apply them to some common situations. Fig. 4 shows the sequence of events when a window is activated and brought to the front. Look at the steps involved and note the roles of the update and activation events, and the formation of the update region.

There are some fine points of handling the scroll bars and grow icon that we will discuss next month, when we cover controls and textEdit.

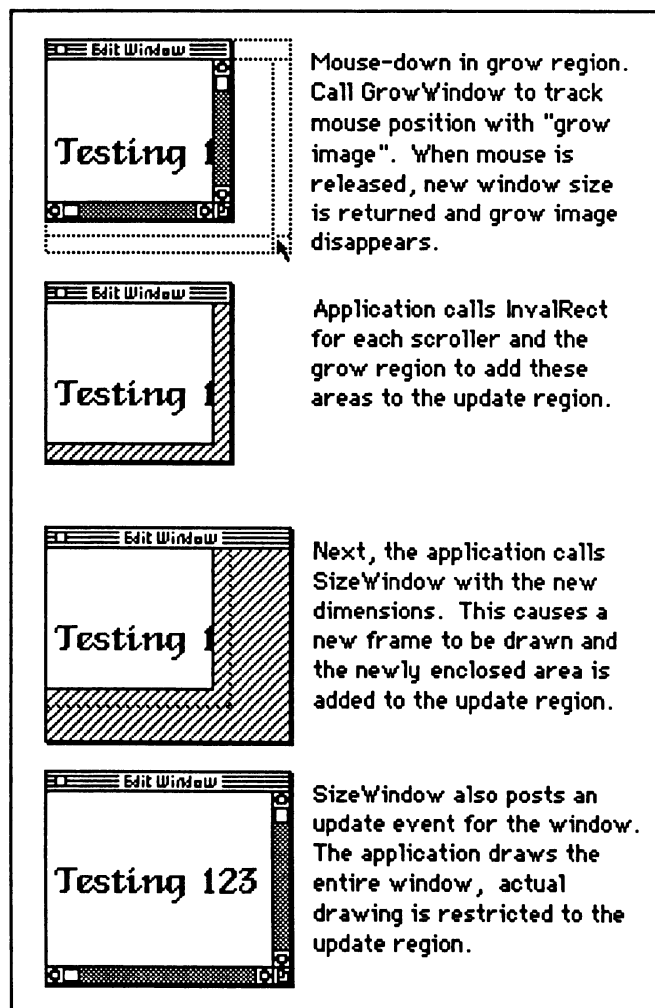


**Fig. 6 Re-sizing a window**

The next illustration, fig. 5, shows what happens when a window which is partially off-screen is dragged back on to the screen. It was this case which turned the light on for me personally.

Interestingly enough, no update event is generated for a window which is dragged from one place to another on the screen. The window manager has a complete image of the window already on the screen, so it simply "bit-blits" (fast image copy) the image from one place to the other.

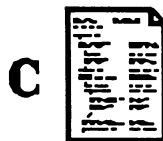
The surprise comes when you drag the window off-screen, release it, then drag it back on. Now part of the window must be reconstructed as if it was being drawn for the first time. An update event is posted and the application must redraw the part that was off the screen.

The last "movie", fig. 6, shows what happens when a window is resized. No activation events are posted for this action. Note particularly the manual invalidation of the scroll bars by calling InvalidateRect() for each. This is needed because the window manager has no idea that the controls are going to be moved, invalidating the area covered by them. Next month, controls and Text Edit!



# C Workshop

## Text Edit & Scrolling Dynamics



Bob Denny  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 6

### Windows for Text Editing

One of the most common uses of windows in Macintosh applications is for manipulation of text. The user interface for text editing is one of the cornerstones of Mac technology.

This month's *C Workshop* deals with implementing editing windows. We'll bring together much of what was presented in previous columns, and add new information on TextEdit's services. We'll also show some specifics on using the Control Manager to handle scroll bars, and their use in scrolling our view of text managed by TextEdit.

One rather important issue that will not be covered is file I/O. It's important to realize that TextEdit manages operations on text in memory. If you want to work with text in a file, you must provide the services for getting text into and out of memory in manageable chunks for TextEdit to work with.

### TextEdit Power

TextEdit is one of the most powerful of Macintosh's system service packages. The Mac designers realized that editing text is probably the most pervasive user operation of all, and that standardizing the editing interface would make the Mac easy to use. Therefore, they have provided us with a remarkably flexible and user-friendly editing package.

TextEdit consists of a set of system services for displaying and manipulating text. Text is stored in coded form in a linear array hooked to a master data structure called a TEREc. The TEREc is to TextEdit as the WindowRecord is to the Window Manager.

The display services of TextEdit handle drawing of text in the window, line breaking with optional word wrap, blinking the caret and showing selected ranges of text. The manipulating services handle text selection, insertion, deletion, and the scrap services of copy, cut and paste, operating on the actual text stored in memory and on the clipboard. It is a good idea to keep the concepts of display and manipulation separate in your mind.

TextEdit is reasonably documented in *Inside Macintosh*. It can be difficult to keep an overall perspective on things, though, when an editing window involves so many of Macintosh's services and their data structures. Figure 1 shows a bird's eye view of most of the data structures involved in an editing window.

### Using TextEdit: Statics

There are three main things to get set up in preparation for using TextEdit with a window. First, you must create a TEREc with a filled-in handle to the array where the text will be stored. Next, you must define two very important rectangles, the ViewRect and the DestRect. First, let's look at the TEREc. As usual, its structure will tell us some

important things about TextEdit. The names of the structure members have been taken directly from the Lisa Pascal interface to TextEdit in the file TOOLINTF.TEXT.

The meaning of most of the fields is shown in *IM*. We'll document some of the others, though you won't normally use them. "selPoint" is the mouseLoc of the current selection point. "active" is a boolean telling whether the related window is active. This is used to control caret blinking and highlighting of selected text during window updates. "wordBreak" is a pointer to the routine that calculates word breaks (this has interesting possibilities). "clickLoop" is a pointer to the routine called when the mouse is clicked in text. It handles tracking the mouse and highlighting the selected text on the fly while the mouse is pressed. "clickTime" and "clickLoc" mark the place and time where & when the mouse was first clicked. Presumably, they are the property of the clickLoop routine. "recalBack" is a boolean that controls whether line breaks will be continuously recalculated in background or not and "recalLines" is a boolean that indicates whether a recalculation is under way. Finally, "caretState" indicates whether the caret is on or not, and "caretTime" contains the time for the next caret blink.

```
struct TEREc
{
 Rect destRect;
 Rect viewRect;
 Rect selRect;
 short lineHeight;
 short firstBL;
 Point selPoint;
 short selStart;
 short selEnd;
 short active;
 long wordBreak;
 long clickLoop;
 long clickTime;
 short clickLoc;
 long caretTime;
 short caretState;
 short just;
 short TElength;
 Handle hText;
 short recalBack;
 short recalLines;
 short clickStuff;
 short crOnly;
 short txFont;
 Style txFace;
 short txMode;
 short txSize;
 GrafPtr inPort;
```

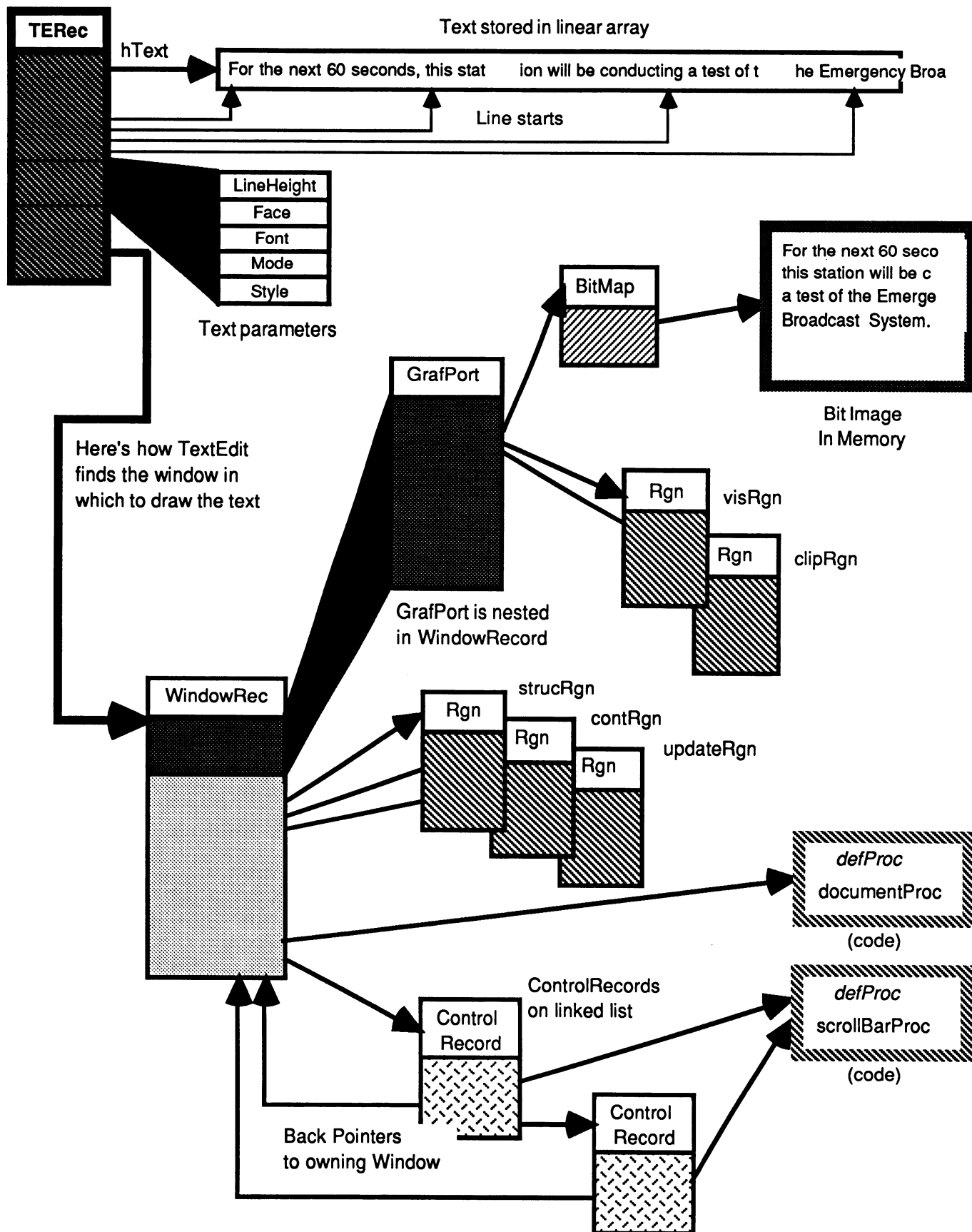


**Fig. 1 How It All Relates**

```
Ptr highHook;
Ptr caretHook;
short nLines;
short lineStarts[1];
};
```

Before doing anything, your application must call `TEInit()`. This allocates a single common scrap area for use by `TextEdit`.

To allocate a `TERec` and a text array for a given window, call `SetPort()` for that window, then call `TENew()`, which returns a handle to the `TERec`. `TENew()` doesn't take a `windowPtr` or a `grafPtr`, rather, it works on the current port, hence the need to call `SetPort()` before `TENew()`.

The two parameters to `TENew` are the `destRect` and the `viewRect`. These two rectangles determine the line width and displayed portion of text. Remember the text is stored in a linear array hooked to the `TERec`. Refer again to Fig. 1. Notice that the `TERec` contains a set of pointers (actually offsets) to the places in the text where line breaks occur. How are these calculated?

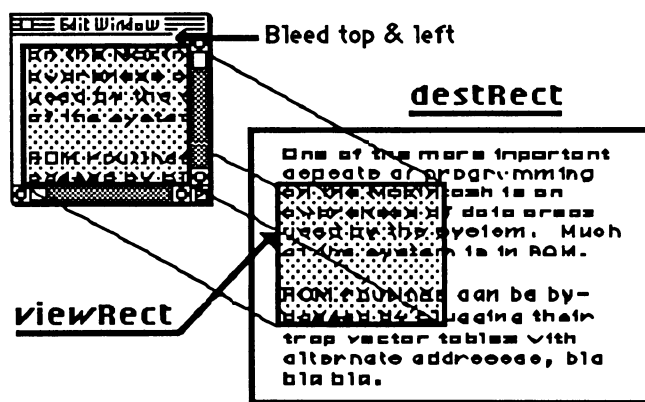


**Fig. 2 Display Mapping**

Figure 2 above shows the relationship between the two rectangles. You can look at the `destRect` as the dimensions of the "chalkboard" where the text is to be displayed. `TextEdit` automatically measures the length of the text in the storage array (given its font parameters) and calculates the location of the line breaks, normally employing word breaks instead of character breaks. The `destRect` is specified in the coordinate system of the owning window's `grafPort`. Don't forget this, it has important implications. In Figure 2, the top left corner of the `destRect` has negative v & h coordinate values.

The `viewRect` specifies the portion of the text that is actually visible. It is sort of a "clip" rectangle for the text display. Why the distinction? Consider the case of displaying text as it would print on 8-inch wide paper. The `destRect` would have a width of 576 pixels (72 x 8). This sets the line breaks for the paper in use.

But the window itself can be resized and scrolled around on the 8-inch wide "chalkboard". The `viewRect` is used to tell `TextEdit` the dimensions and location in the `grafPort` of the "view-port" to the text display being constructed on the chalkboard.

Normally, the `viewRect` is set to completely enclose the content region of the window, less the scroll bar and size box areas. It too is specified in `grafPort` coordinates.

One final word about setting up a window for text editing. It's a good idea to provide a 3 or 4 pixel "bleed" on the top and left of the page image. There are two schools of thought on this. One says that the bleed should be present regardless of the location of the `viewRect` in the `destRect`. The other (to which I subscribe) says that the bleed should be present only if the `destRect` is at the left and/or top of the window port. The latter makes it visually easy to tell whether there is text to the left and above the `viewRect` by showing fragments of letters at the window's edge.

### Using TextEdit: Dynamics

Once you have set up a window for editing, you can start manipulating and displaying text. Keep in mind the following facts about `TextEdit`:

1. It works only on memory images of text. No file I/O is provided.
2. It does not support multiple of character fonts, styles or sizes in a single environment.

The simplest operation is **insertion** of text. This can be done one character at a time with `TEKey()` and *en masse* with `TEinsert()`. It is most common to use `TEKey()` in response to a keypress event. `TextEdit` sticks the new text into the linear array, recalculates the downstream line breaks, then posts an update event for the window, if needed.

Simple **deletion** is also easy. To delete the character to the left of the current insertion point or to delete the current selection, call `TEKey()` with the "backspace" key code. To delete the current selection (if any), call `TEDelete()`. `TextEdit` removes the deleted text from the linear array, closing up the space, recalculates the the downstream line breaks, then posts an update event for the window, if needed.

Mass operations of **cut**, **copy** and **paste** are hardly more difficult. Normally, you should provide both menu options and "menu keys" to trigger these operations, in accordance with the Macintosh interface guidelines. Call `TECut()`, `TECopy()` or `TEPaste()` to perform the operation. If there is no text currently selected, these functions are no-ops. Otherwise, the indicated operation is performed, including transfers to or from the scrap allocated by `TEInit()`, and the already described manipulations of the linear text array and display.

How do we **select text** with the mouse? Recall that the `TERec` contains a pointer to the "clickLoop" routine, the routine responsible for tracking the mouse while it is kept pressed and highlighting selected text on the fly as the mouse is moved. How does this process get started? By calling `TEClick()`.

So whenever your application detects a mouse press inside the `viewRect` of an editing window, it should call `TEClick()`. You can determine whether the mouse was pressed in the `viewRect` by calling the QuickDraw `ptInRect()` service with the clicked mouse location and the `viewRect`. If it returns

TRUE, then call `TEClick()`. The "clickLoop" routine is entered and continues to execute until the mouse is released, at which time control is returned to the statement following the call to `TEClick()`.

To implement the "shift-click" method of extended selection, you must determine whether the shift key was pressed at the time the mouse was clicked and pass TRUE for the "ext" parameter to the `TEClick()` call. For more information, see [IM](#).

To scroll the text display in the `viewRect`, simply call `TEScroll()`. This function takes as parameters the distance (in pixels) to scroll horizontally and vertically. A zero means no scrolling in that direction.

Now let's test your understanding of TextEdit's coordinate system. The first person write in with the correct answer to the following question will receive a free copy of the MacTutor "source disk":

### QUESTION:

Suppose there was no `TEScroll()` function. How would you scroll the displayed text? Provide a C language implementation of `TEScroll()`:

```
tescroll(dh, dv, th)
short int dh; /* Horiz amount */
short int dv; /* Vert amount */
TEHandle th; /* Handle to TEREc */
{
 (... your code ...)
}
```

You must answer the question completely, i.e., furnish a complete implementation of `TEScroll()`, given the parameters `dh` and `dv` whose sense of positive are text motion of right and down, respectively.

There are several other TextEdit routines which can be used to perform special (unusual) operations. Also, you can make changes to some of the `TERec`'s fields yourself. In either case, take care to call `TEUpdate()` to force display changes, and/or `TECalcText()` to force recalculation of line breaks as required.

### Using Scroll Bars

Windows used for text editing generally have at least one scroll bar present (the vertical scroller). A scroll bar is a Macintosh "control", supported by the services provided by the Control Manager. A general discussion of the control manager is beyond the scope of this article. We'll restrict coverage to that necessary to use scroll bars. Actually, the scroll bar is the most complex of the "standard" controls, so once you understand it, you'll have no trouble going back to understand the other controls.

Each scroll bar has associated with it a `ControlRecord`, which is a data structure containing the information needed by the Control manager to handle the control. Most importantly, each control has a value at a given instant. The whole purpose of a control is to visually manipulate the value of the control. Keep this in mind at all times!

Refer to Figure 1. Note that the `ControlRecords` for the vertical and horizontal scrollers are linked (by handles) off the `WindowRecord`. Also, they contain back pointers to the `Window Record`.

The appearance and structural makeup of a particular kind of control is determined by a "defProc", a control definition procedure. We discussed defProcs last month in reference to windows. When you create a control, its "flavor" is determined by the defProc. The control manager is a collection of general purpose routines; the specifics are handled by calling the defProc.

Scroll bars are handled by a defProc named "scrollBarProc". This defProc is fairly smart. For example, when creating a scroller, you pass the location and dimensions (in window local coordinates) of the enclosing rectangle. The defProc is smart enough to look at the dimensions and automatically determine if the scroller is to be a horizontal or vertical flavor (I've never tried to pass a square rect).

The defProc for the "document window" (`documentProc`) assumes that the scrollers will occupy 16-pixel strips along the right and bottom edges, with a one-pixel overlap of the window's border. The grow image assumes this, and the size and location given to the size box assumes this. The scrollers are in the window's content region. We detailed this in a previous edition of MacTutor.

When you detect a `MouseDown` event in the content region of a document window, call `findControl()`. If the click was in a control, the handle to the control is returned along with a "part code" indicating what part of the control the mouse was clicked in. These part codes are shown symbolically in Figure 3. We'll deal with actions to be performed later.

A frequently misunderstood area involves the display states that scrollers can take. When an edit window is inactive, the scrollers are invisible: only the scroller outlines drawn by the window's defProc show. There are three visible states, however, as shown in Figure 4 (next page).

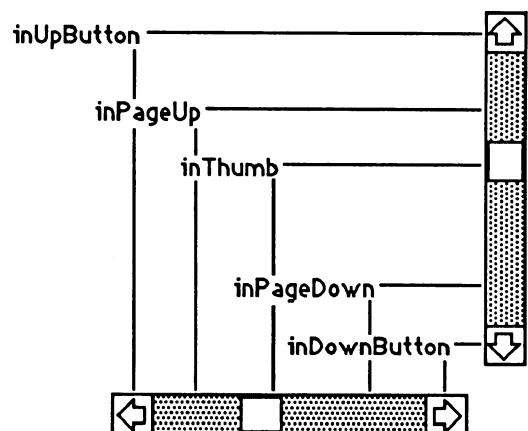


**Fig. 3 Scroll Bar Parts**



You'll probably never have occasion to highlight parts of a scroller yourself. However, you may want to make the scroller "inactive" if there is nothing beyond the limits of the viewRect, if there is no reason to scroll. To make the scroller look inactive, call the highlighting function `HiLiteControl()` with a "part code" of 254 or 255. Using 254 leaves the scroller sensitive to mouse presses, and calling `FindControl()`, 255 will not.

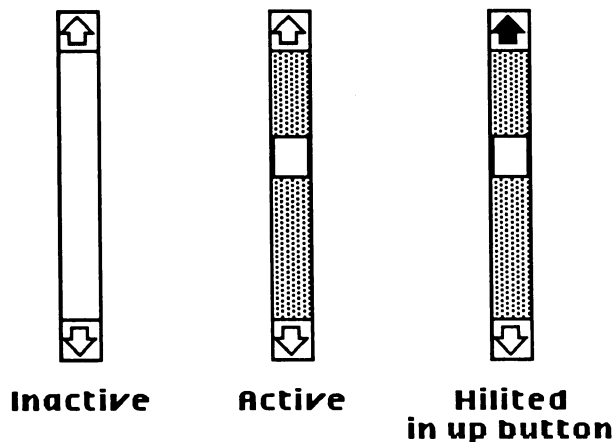


**Fig. 4 Control Display States**

#### Putting It Together: Creating a Document Window

Here is some C code for creating a document window. It assumes that there is a window template resource, just for simplicity here. You can substitute `NewWindow()` for `GetNewWindow()`:

```
/*
 * EDIT_WINDOW() - Create an "edit window"
 *
 * Inputs:
 * dest:-> destination rect
 *
 * Outputs:
 * Fills in vs_handle and hs_handle
 * with scroller handles.
 * Fills in te_handle with handle to
 * TERec.
 * Returns the WindowPtr
 *
 * WARNING: Untested extract.
 */
WindowPtr edit_window(dest)
Rect *dest;
{
 Rect view_rect;
 Rect bounds_rect;
 Rect dest_rect;
 Rect drag_rect;
```

```
Rect grow_rect;
Rect *pr;
WindowPtr wp;

/*
 * Create window record on heap from
 * resource ID = "WINDOW_ID". We use
 * some vanilla values for the window
 * drag and grow rects.
 */
wp = GetNewWindow(WINDOW_ID,0,-1);
SetWTitle(wp,"013Edit Window");
SetRect(&drag_rect,4,24,508,338);
SetRect(&grow_rect,100,60,512,302);
SetPort(wp);

/*
 * Set up TextEdit rects. Make the
 * destRect fixed here for example.
 * Co-locate the viewRect's top right
 * corner with the destRect's for 4
 * pixel bleed. Leave room for scrollers
 */
pr = &(wp->portRect);
destRect.top = destRect.left = 4;
destRect.bottom = 1000;
destRect.right = 500;
viewRect.top = destRect.top;
viewRect.left = destRect.left;
viewRect.bottom = pr->bottom - 15;
viewRect.right = pr->bottom - 15;

/*
 * Now create the TERec & text area
 */
te_handle = TENew(&dest_rect,
 &view_rect);

/*
 * Calculate the vertical scroller rect
 * and add the scroller to the window.
 */
bounds_rect.top = pr->top - 1;
bounds_rect.left = pr->right - 15;
bounds_rect.bottom = pr->bottom - 14;
bounds_rect.right = pr->right + 1;
vs_handle = NewControl(dwp->wp,
 &bounds_rect,
 "", TRUE, dest_rect.top,
 dest_rect.top, dest_rect.bottom,
 scrollBarProc, 1);
ValidRect(&bounds_rect);

/*
 * Same for the horizontal scroller
 */
bounds_rect.top = pr->bottom - 15;
bounds_rect.left = pr->left - 1;
bounds_rect.bottom = pr->bottom + 1;
bounds_rect.right = pr->right - 14;
hs_handle = NewControl(dwp->wp,
```

```

 &bounds_rect,
 "", TRUE, dest_rect.left,
 dest_rect.left, dest_rect.right
 scrollBarProc, 1);
ValidRect(&bounds_rect);

/*
 * Finally, draw in the "grow icon" &
 * return the window pointer
 */
DrawGrowIcon(wp);
return(wp);
}

```

There are some fine points to be aware of here. The scrollers are created with bounds rectangles which are calculated from the window's portRect. The viewRect for TextEdit is also calculated from the window's portRect. This means that the window template resource can be any reasonable size and this routine will still work.

NewControl() not only initializes the ControlRecord, it also draws the control (set to its initial value). The calls to ValidRect() prevent unnecessary drawing in response to an update event. The call to SetWTitle() passes a "Pascal" (counted) string generated the quick way.

Finally, note that the scrollers are initialized with values drawn from the TextEdit destRect. More about this later.

### Event Handling:MouseDown

If your application detects a mousedown event in the content region of a document window, it must determine if the click was in the viewRect, one of the scrollers, the grow region or a "bleed" area. I suggest you call PtInRect() first, with the viewRect. If it returns something other than FALSE (0), then the click was in a TextEdit-controlled region. If not, call FindControl(). If it returns something other than NULL (0), it was in a control. If not, the click was in a bleed area and can be ignored. Here is some C code for handling a content-area click.

```

/*
 * content_click()
 *
 * Event record "event" is global. Shift
 * click selection not supported.
 *
 * WARNING: Untested extract.
 */
content_click()
{
 ControlHandle ch;
 unsigned short part;
 int in_text;
 /*
 * Declare assembler action
 * routines for TrackControl().
 */
 int scroll_up();

```

```

int scroll_down();

/*
 * Convert event loc to current grafPort
 * local coordinates & test if in text.
 */
GlobalToLocal(&Event.where);
in_text = PtInRect(&event.Where,
 &((*te_handle)->viewRect));

/*
 * If in text, enter the "click loop"
 * otherwise, handle scroller or bleed
 * click.
 */
if(in_text)
 TEClick(&Event.where, 0, te_handle);
else
{
 part =
 (short)FindControl(&Event.where,
 (*te_handle)->inPort, &ch);
 switch(part)
 {
 case inUpButton:
 TrackControl(ch, &Event.where,
 scroll_up);
 break;

 case inDownButton:
 TrackControl(ch, &Event.where,
 scroll_down);
 break;

 case inPageUp:
 page_scroll(part, ch, -1);
 break;

 case inPageDown:
 page_scroll(part, ch, 1);
 break;

 case inThumb:
 TrackControl(ch, &Event.where,
 0);
 edit_scroll();

 default: /* In bleed */
 }
}
}

```

### How to Handle Scrolling The Easy Way

There are several mystery functions in the code given above. First, and most important, is edit\_scroll(). It is the basis for all of the other mystery functions. But before we go into the mechanics of "easy scrolling", let's look at the basic method.

The key to the method is the relationship between the scroll bar's "control value" and the position of the displayed text. Recall that TEScroll takes *relative* horizontal and vertical scroll distances, movement amounts. To nail things down, we first define a coordinate pair (a Point) which will contain the distance in pixels that we have scrolled from the original top left. We'll call this `te_origin`.

Now for the main thing. The scroller "values" are used to control the values of `te_origin`'s vertical and horizontal coordinates. The range of values for the horizontal scroller is `destRect.left` to `destRect.right`. Likewise, the range of values for the vertical scroller is `destRect.top` to `destRect.bottom`. Think about that one...

Any time a scroller's value changes, the new value is compared with the corresponding value of `te_origin`. The difference is used to call TEScroll() to do the actual scrolling, then the new value replaces the old value in `te_origin`. Here is the C code to implement this basic scrolling function. (Please don't criticize this for tightness - it's just an example):

```
/*
 * edit_scroll()
 *
 * Inputs:
 * Current value of te_origin
 * Current values of the scrollers
 *
 * Outputs:
 * te_origin is updated
 * Window is scrolled to new loc
 *
 * WARNING - untested extract
 */
edit_scroll()
{
 short int nh, nv, dh, dv;

 nh = GetCtlValue(hs_handle);
 if(nh < 4) nh = 4;
 dh = te_origin.h - nh;
 te_origin.h -= dh;

 nv = GetCtlValue(vs_handle);
 if(nv < 4) nv = 4;
 dv = te_origin.v - nv;
 te_origin.v -= dv;

 #TEScroll(dh, dv, te_handle);
}
```

This function is all we need to handle the "inThumb" case of scrolling in the `content_click()` routine above. A click in the thumb causes us to call `TrackControl()`, which keeps control until the mouse is released. At that point we get control back and immediately call `edit_scroll()`, which reads the control values and scrolls as required.

The next case to handle is clicking in the up and down buttons. For the vertical scroller, the proper action is to repeatedly scroll one "lineHeight" until the mouse is released.

For the horizontal scroller, you might choose a "jump" scroll of 1/8 of the window width, or something like that.

The proper way to accomplish this repeated scrolling action is to do so with an action routine supplied to the `TrackControl()` function. These are the mystery `scroll_up()` and `scroll_down()` functions used in `content_click()` above. Unfortunately, they cannot be written in C. This is because they are "called back" from the `TrackControl()` function, and the callback uses the LisaPascal linkage interface. Here is the assembly code for the vertical scroller's `scroll_up()`:

```
;
; scroll_up()
;
; TrackControl callback action routine for
; vertical scroller's scroll-up one line
; function. Called when up-arrow is
; pressed.
;
; Inputs:
; 4(sp) Part code (short int)
; 6(sp) ControlHandle (32-bit address)
;
scroll_up:
 Link A6,#0 ; Link env.
 Move.W 8(A6),D0 ; D0 = part code (W)
 Beq @1 ; (not in arrow now)
 Move.L 10(A6),-(SP) ; Save handle
 Clr.W -(sp) ; Gets control value
 Move.L 10(A6),-(SP) ; Pass handle
 _GetCtlValue ; Get the value
 ;-----
 Jsr get_lh ; D0 = line height
 Sub.W D0,(SP) ; (SP) = new ctl val
 Bge @0 ; (OK, its positive)
 Clr.W (SP) ; Limit at 0
 @0:
 _SetCtlValue ; Set new value
 Jsr edit_scroll ; Now scroll
 @1:
 Unlk a6 ; Pascal exit
 Move.L (SP)+,A0
 Addq #6,SP
 Jmp (A0)
```

Note the statements between the dashed lines. This is the only difference between the horizontal and vertical scroller's callback routines for up and down. The difference is in the amount and direction of change to the scroller's value in (SP). The example above makes use of a mystery routine `get_lh()`, which reads the line height from the `TERec`. It also calls our `edit_scroll()` routine to scroll the window once the scroller's value has been changed. The call to `_SetCtlValue` changes the value of the scroller and moves the thumb box accordingly.

The final thing we need is a routine for page scrolling. This time we won't use `TrackControl()`. Since we are not being called back, the routine can be written in C, and can take a "direction" parameter, so one routine is all that is needed.

The Macintosh Interface Guidelines specify a particular

type of "action" that we must implement in the page\_scroll() routine. That is, the page scroll must be done at least once, and repeated as long as the mouse is held down and the point is inside the original page scroll area. If the mouse is moved out of that page scroll area, page scrolling should stop, until it is moved back into the area, at which time page scrolling should resume.

The following page scrolling routine does this and scrolls different amounts for vertical and horizontal scrollers. See if you can figure out what is happening:

```
/*
 * page_scroll()
 *
 * Inputs:
 * part Part code where first clicked
 * ch ControlHandle
 * dir Direction: -1=Up 1=Down
 *
 * Locks the TEREc temporarily
 */
page_scroll(part, ch, dir)
short part;
ControlHandle ch;
short dir;
{
 Point cur_pt;
 short amount;
 Rect *vr;

 HLock(te_handle); /* Lock & deref */

 vr = &((*te_handle)->viewRect);
 if(ch = vs_handle)
 amount = vr_bottom - vr->top -
 (short)get_lh();
 else
 amount = (vr->right - vr->left)/2;

 amount *= dir;

 /*
 * Act in accordance with Interface
 * Guidelines.
 */
 do {
 GetMouse(&cur_pt);
 if((short)TestControl(ch, &cur_pt)
 != part)
 continue;
 SetCtlValue(ch, GetCtlValue() +
 amount);
 edit_scroll();
 } while(StillDown());

 HUnlock(ch); /* Release TEREc */
}
```

That's about it for scrolling. There's a lot of hidden information in the routines given. If you'll take the time to

understand them (and the associated toolbox traps), you will have a great start at one of the most complex areas of Mac application design.

## Event Handling: Updates

Handling update events for document windows is surprisingly simple. TextEdit and the Control Manager do most of the work. Update events call for redrawing the window's contents. BeginUpdate() and EndUpdate() restrict the actual drawing to that which is necessary. Here is an example of an update event handler for document window:

```
/*
 * Upd_wind() - Update document window
 *
 * Inputs:
 * wp = WindowPointer
 * th = TEHandle
 */
upd_wind(wp, th)
WindowPtr wp;
TEHandle th;
{
 BeginUpdate();
 SetPort(wp);

 DrawGrowIcon(wp);
 DrawControls(wp);
 TEUpdate(&((*wp->visRgn))->rgnBBox,
 th);

 EndUpdate();
}
```

That's it. Notice the rectangle passed to TEUpdate(). Instead of drawing the entire ViewRect, this method uses the visRgn, which has been temporarily hacked to be the update region by BeginUpdate(). This is one case where it is easy to manually restrict drawing to the update region. Doing this can result in substantial speed improvements for TEUpdate. Notice how easy it is to redraw the controls.

## Event Handling: Activate and Deactivate

Activation (and deactivation) events are also quite easy to handle for document windows. Activation calls for changing the "look" of the controls and text, and turning on the caret or selection range. Actually, for scrollers, activation means "showing" them, deactivation means "hiding" them.

There is no "ShowControls()" function, so we "show" each one by tracing down the linked list. This method makes it unnecessary to pass control handles:

```
/*
 * act_wind()/deact_wind()
 *
```

```
* Activate event handlers
```

```
*/
```

```
act_wind(wp)
WindowPtr wp;
{
 Handle ch;

 SetPort(wp);

 DrawGrowIcon(wp);
 TEActivate(dp->te_handle)
 ch = ((WindowPeek)(wp))>->controlList;
 while(ch != NULL)
 {
 ch =
 (*((ControlHandle)(ch)))>->nextControl
 ShowControl(ch);
 }
}

deact_wind(wp)
WindowPtr wp;
{
 Handle ch;

 SetPort(wp);

 DrawGrowIcon(wp);
 TEDeactivate(dp->te_handle)
 ch = ((WindowPeek)(wp))>->controlList;
 while(ch != NULL)
 {
 ch =
 (*((ControlHandle)(ch)))>->nextControl
 HideControl(ch);
 }
}
```

Note that each routine calls DrawGrowIcon(). That function knows whether the window is active or inactive, and fills in the grow box appropriately. In other words, DrawGrowIcon() for an inactive window blanks out the grow box.

### Final Words

As promised last month, I have used TextEdit and controls to tie together the information given in the last two issues, and to illustrate applications of basic Mac principles. The routines shown in this month's column are **untested examples!** In most cases, the code was lifted out of working applications, then simplified for ease of understanding. The algorithms should be correct, the detail may not be.

Due to the size of even a simple C application, we won't be publishing sources for complete applications in MacTutor. Rather, we'll make them available on the MacTutor source disks.

I plan to devote next month to a random collection of operating system hacks, maybe an interface to Standard File or a C implementation of sprintf() for those of us who abhor Unix libraries for Mac applications.

### Heinrich "Benchmark" Revisited

In the February 1985 edition of MacTutor, we published this program by Mr. Robert Heinrich of Boca Raton, FL:

```
main()
{
 union u_storage{
 long a_long;
 struct T_0000{
 short a_short;
 short b_short;
 }S_0000;
 }storage;
 storage.a_long = 6;
 printf("na_short = %d",
 storage.S_0000.a_short);
 printf("nb_short = %d",
 storage.S_0000.b_short);
 printf("\n");
}
```

He was looking for the answers "a\_short = 0" and "b\_short = 6". The nature of this program compels me to make some comments.

The union maps 2 16-bit words over a 32-bit longword. The order of addressing the two words in the longword is system-dependent.

The 68000 stores the least significant byte of a word at address  $n$  and the most significant half at address  $n+1$ . Likewise, it stores the most significant half of a longword at address  $n$  and the least significant half at address  $n+2$ . It is this latter property that the program uncovers.

Other machines (such as the DEC PDP-11 and VAX systems) store words and bytes in the reverse order. There are sound reasons for each way, and I'll not argue either point.

What this program has brought up is the "discussion" of whether or not the C language should hide such machine dependencies from the programmer.

Firstly, the current C languages do not hide machine dependencies. The new ANSI standard does not call for machine independence either.

Some people feel that C is a high-level language and therefore a C program written for machine X should run on any other machine (except for OS specific, of course).

I couldn't disagree more. C is a "system implementation" language. The whole idea of C is to amplify the programmer's productivity and enhance maintainability by providing a viable alternative to assembly language.

If C compilers scrambled the addressing of struct members, it would make the language nearly impossible to use for system programming. In fact, there are many people who feel that automatic padding to insure correct alignment of structure members is not good. The Mac C compiler has an option to control structure padding.

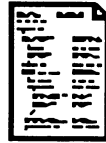
C provides a well-defined access to low-level machine specifics. I don't want that to change.



# C Workshop

## Custom File Type Dialog Box

C



Bob Denny  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 7

I have often wanted to have the ability to put up a dialog box like that of "Standard File", where the user could select from a list of strings.

Implementing such a function requires a combination of most of the Macintosh technology we have covered in the past several months, plus the services of the Dialog Manager. And not just the usual dialog services: we'll need to implement a *filter function* to manage the selection box and scroll bar(s) within the dialog.

This month's *C Workshop* makes up for past columns which showed relatively few C programming examples. Instead of explaining the theory of using the dialog manager and its filter function hooks, I am instead submitting a complete C function that can be used with most compilers which implements a single-selection dialog similar to that of Standard File.

The appearance of the dialog is completely controlled by resource data, including the dimensions of the selection box. Scroll bars are provided in both vertical and horizontal axes, and automatically size to the selection box dimensions. This particular example was used in a utility to perform name lookups on AppleTalk using the Name Binding Protocol (NBP), and allow selection of a particular object by name.

To use the function `sel_dialog()`, call it with 3 arguments: a "call-back" function name (explained below) and the address and size of a buffer to receive a copy of the selected string. Four buttons control its operation.

"Lookup" causes the "call-back" function to be called repeatedly for pointers to P-strings to put into the selection window. This action continues until the call-back function returns NULL. The strings do not have to be kept around after being passed to `sel_dialog()`, but are stored in the TERec's linear text array. The Lookup button may be pressed repeatedly for many fill-in cycles.

If any of the strings are too long to fit in the selection box horizontally, the horizontal scroller is turned on and ranged for the longest string. It continues to be ranged for the longest string as additional strings are added thereafter. Likewise, the vertical scroller is turned on and continually ranged if there are more strings than will fit vertically in the selection box.

After a lookup cycle, the strings in the selector box are armed for mouse clicks and are highlighted when clicked. If any strings are highlighted, the "Accept" button is turned on.

Pressing "Accept" causes the selected string to be copied into the buffer supplied as an argument. "Cancel" causes `sel_dialog()` to return with a null string in the caller's buffer. In either case the dialog and all associated data in memory is released, and a return to the caller is made.

"Clear" causes the selector box to be cleared; all strings are

erased and the scrollers are reset and turned off. This might be useful in preparation for a new lookup process.

### Resources Control Appearance

`Sel_dialog()` is written to be controlled by a set of resources which completely determine the dialog's layout, including the location and titles of the buttons and the location and dimensions of the selection window. The example here is very similar to the dialog used by Standard File, but it needn't be. Just observe the correlation between the buttons' DITL item numbers and their functions. Obviously, you can change the code to eliminate buttons, but you can do so as well by "placing" the buttons off- screen without loss of generality of the function.

Here is the RMAKER source for the example shown on the next pages, followed by some pictures of the dialog and the code for `sel_dialog()`. Note the homebrew 'RECT' resource which sets the location and dimensions of the selection box.

TYPE DLOG  
,256

72 72 224 420  
Invisible NoGoAway  
1  
0  
256

TYPE DITL  
,256  
7

button  
28 152 46 232  
Accept

button  
59 256 77 336  
Lookup

button  
90 256 108 336  
Clear

staticText Disabled  
28 254 46 344  
NBP Lookup

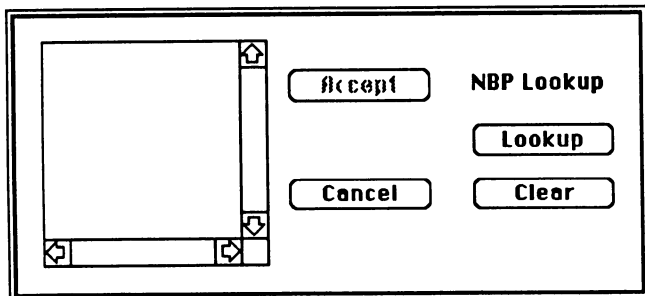
\*Selector box Rect  
TYPE RECT = GNRL  
,256

```
.l
11 12 125 125

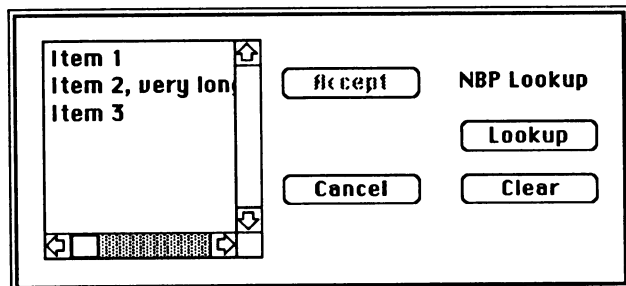
button
90 152 108 232
Cancel
```

By any measure, this is an "advanced" programming example. Little explanation accompanies the code. I highly recommend you review the Dialog Manager section of *Inside Macintosh* before working through this example. Note how the filter function and modal dialog loops interact to provide the user interface with just a simple function call from the client application.

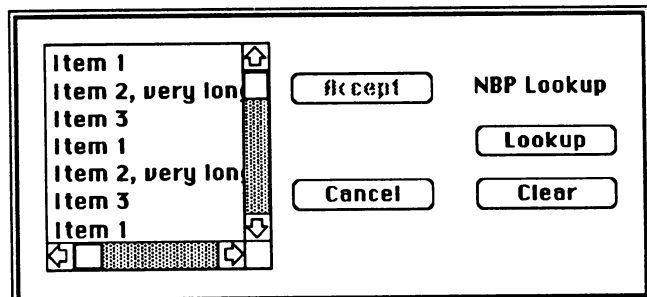
The dialog box looks like this when it first appears:



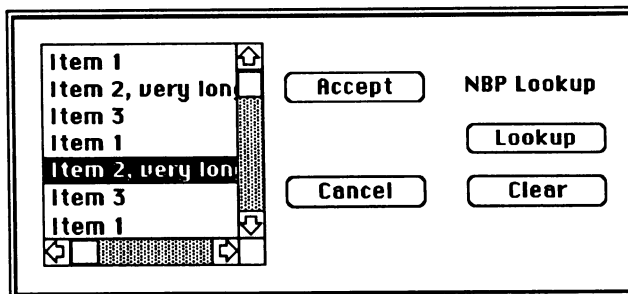
If the "Lookup" button is pressed, your routine gets called repeatedly for P-strings to be inserted in the list shown in the window, until it returns NULL. If one of the strings is longer than will fit in the window, the horizontal scroller will be turned on and ranged appropriately, like this:



If your routine returns enough strings to fill up the window vertically, the vertical scroller will be turned on, like this:



After your call-back function has finished, the mouse becomes active and may be used to push buttons and select text. If you click in one of the items in the select window, it will be highlighted and the "Accept" button will be activated like this:



The C code which makes up the rest of this article implements this general purpose selection dialog. Macintosh tool-box calls are italicized for emphasis. The assembly routines use the linkage conventions of Consulair Mac C, and will need minor changes for other C implementations. Other than these differences, the code below should be easy to use under any of the C systems available for the Mac. There are no "library" routines used at all, except those in the Macintosh ROM.

```
/* Resource IDs */
#define DLOG_ID 256 /* Dialog window itself */
#define BOX_ID 256 /* Selector box Rectangle */

/* DITL item numbers, also part codes */
#define ACCEPT 1 /* The Accept button */
#define CANCEL 2 /* The Cancel button */
#define LOOKUP 3 /* The Lookup button */
#define CLEAR 4 /* The Clear button */
#define TITLE 5 /* static title (NBP Lookup) */

/* Local static variables */
static Rect box_rect; /* Box rect (from resource) */
static ControlHandle v_scroll; /* -> V-Scroller */
static ControlHandle h_scroll; /* -> H-Scroller */
static ControlHandle acc_button; /* -> Accept button */
static Rect dest_rect; /* TextEdit's destination rect */
static Rect view_rect; /* Text Edit's view Rect */
static short int vs_offs; /* destRect vertical offset */
static short int hs_offs; /* destRect horiz offset */
static TEHandle hTE; /* Handle to TextEdit record */
static unsigned short line_height; /* Text line height, pixels */
static unsigned short lines_vis; /* No. of whole lines visible */
static unsigned short half_wid; /* half width of box in pixels */
static Point loc_pt; /* Local coord of last mouse click */

/*
 * SEL_DIALOG() - MAIN DIALOG FUNCTION
 *
 * Inputs:
 * P1 procedure to call to add an item. Returns pointer to
 * P-string to add to the items in the box (no newline);
 */
```

```

* Must return NULL when no more to add.
* P2 address of buffer to receive selection
* P3 length (bytes) of selection buffer
*
* Outputs:
* If Accept pressed, selected string is copied as a
* C-string into the P2 buffer, up to P3-1 bytes. If Cancel
* pressed, a null C-string is placed into the P2 buffer
*/
sel_dialog(add_item, sel_buf, buf_len)
char *(*add_item)(); /* User call-back to add item */
char *sel_buf; /* Buffer to receive selection */
short buf_len; /* Length of buffer to receive selection */
{
 Ptr dp; /* Dialog pointer */
 Rect **rh; /* Handle to resource rect */
 Rect scr_rect; /* Scratch rect */
 long il; /* Length of text from user */
 char *cp; /* Ptr for scanning user items */
 short scratch; /* Scratch word */
 unsigned short item_hit; /* Item number hit in dialog */
 unsigned short done; /* Flag indicating he's done */
 unsigned short sw; /* String width, pixels */
 unsigned short cv; /* Control value buffer */
 unsigned short max_wid; /* Max line width pixels */
 unsigned short sel_line; /* Line number of selected line */
 unsigned short sel_start; /* Index of 1st char of sel line */
 unsigned short sel_end; /* Index 1st char after sel line */
 unsigned short sel_len; /* Length of selected text */
 short hce, vce; /* Flags TRUE if scroller enabled */
 int user_Filt(); /* Dialog filter */

 dp = (Ptr)GetNewDialog (DLOG_ID, 0, -1); /* Load dialog */
 SetPort (dp); /* Hook QuickDraw to dialog */
 /*
 * Dim the accept button & save it's handle for later
 * Disable it until something is selected
 */
 GetDItem (dp, ACCEPT, &scratch, &acc_button, &scr_rect);
 HiliteControl (acc_button, 255);
 /* "part" 255 means dim & disable */
 /*
 * Get the box rect as a resource & copy it
 * locally, then release it
 */
 rh = (Rect *)GetResource ('RECT', BOX_ID); /* Get rect */
 box_rect.topLeft.all = (*rh)->topLeft.all;
 box_rect.botRight.all = (*rh)->botRight.all;
 ReleaseResource (rh); /* No need for this now */
 /*
 * Put up the scroll bars. Compute their sizes from the
 * selector box rectangle.
 */
 scr_rect.top = box_rect.top; /* Vert scroll bar on right edge */
 scr_rect.left = box_rect.right - 1; /* Overlap wind box 1 pix */
 scr_rect.bottom = box_rect.bottom;
 scr_rect.right = scr_rect.left + 16; /* Standard 16-pix width */
 v_scroll = (ControlHandle)NewControl (dp, &scr_rect, 0,
 TRUE, 0, 0, 0, scrollBarProc, 0);
 HiliteControl (v_scroll, 255); /* Deactivate for now */
 vce = FALSE;

```

```

scr_rect.top = box_rect.bottom - 1; /* Horiz. scroll bar */
scr_rect.left = box_rect.left; /* Same overlap, width */
scr_rect.bottom = scr_rect.top + 16;
scr_rect.right = box_rect.right;
h_scroll = (ControlHandle)NewControl (dp, &scr_rect, 0,
TRUE, 0, 0, 0, scrollBarProc, 0);
HiliteControl (h_scroll, 255); /* Deactivate for now */
hce = FALSE;

```

```

/*
* Set up a TextEdit record for the items in the box. Leave 3
* pixel bleed on left, none on the
* right, 3 at the top of the destRect. Must never wrap or go
* off bottom. Calculate the number
* of complete lines of text visible in the view_rect.
*/
dest_rect.top = box_rect.top + 3;
dest_rect.left = box_rect.left + 3;
dest_rect.right = 1000; /* room for long text items */
dest_rect.bottom = 2000;
view_rect.top = box_rect.top + 1; /* View area inset 1 pixel */
view_rect.left = box_rect.left + 1;
view_rect.bottom = box_rect.bottom - 1;
view_rect.right = box_rect.right - 1;
hTE = (TEHandle)TENew (&dest_rect, &view_rect);
/* Start up TextEdit on the box */
/*
* Calculate scrolling parameters & initialize vars. This
* makes it possible to control
* everything by changing the box Rect resource.
*/
line_height = (*hTE)->lineHeight; /* Copy line height */
hs_offs = vs_offs = 0; /* Viewing upper left corner */
lines_vis = (view_rect.bottom - view_rect.top) / line_height;
half_wid = (view_rect.right - view_rect.left) / 2;
/*
* Finally - Light up the dialog box all at once
*/
ShowWindow(dp); /* Make it visible suddenly */

/*
* MAIN DIALOG LOOP
*/
done = FALSE; /* Not "done", obviously */
max_wid = 0; /* Longest line (pixels) */
hce = vce = FALSE; /* No scrollers needed now */

while(!done) /* "Done" if Accept or Cancel pressed */
{
 ModalDialog (user_Filt, &item_hit);
 /* Do dialog, return item # hit */

 switch(item_hit) /* Take action from dialog */
 {
 /*
 * CANCEL pressed. Axe everything and return an empty
 * item buffer. Exit the dialog loop * clean up.
 */
 case CANCEL: /* Forget it, no returned item */
 done = TRUE; /* We're "done" */
 sel_buf[0] = '\0'; /* NULL string */

```



```

break;

/*
 * ACCEPT pressed. Grab the text of the selected line
 * and return it to the caller.
 * Exit the dialog loop.
 */
case ACCEPT:
 done = TRUE; /* We're "done", all right */
 sel_len = sel_end - sel_start; /* Length (bytes) */
 if(sel_len > buf_len) /* Don't overrun buffer! */
 sel_len = buf_len;
 BlockMove (((char *) ((*hTE)->hText))[sel_start],
sel_buf, sel_len);
 break;

/*
 * LOOKUP pressed. This is the most complex:
 *
 * 1. Deselect currently selected item, if any, and
 dim Accept button.
 * 2. Repeatedly do call-back for P-strings to add to
 items in selection box.
 * Stop when call-back returns NULL. Keep track of
 size of longest item
 * (in pixels).
 * 3. When all new items have been added, turn on
 required scrollers. H-scroller
 * is needed if longest item exceeds view_rect width.
 V-scroller is needed if
 * total number of items exceeds the number of lines
 visible in the box.
 * 4. For each live scroller, adjust it's range to just
 cover the live area. For the
 *
 * V-scroller, set it's max so that the last item will be at
 the bottom of the
 * box when the control is at max. For the H-scroller,
 set it's max so that
 * the longest item's last character will be at the right
 edge of the box when
 * the control is at max.
 *
 * NOTE: The Lookup button may be repeatedly pressed.
 */
case LOOKUP: /* Call user proc till we get NULL */
 TEDeactivate (hTE); /* Hide selection */
 TESetSelect (32767, 32767, hTE); /* Deselect */
 sel_start = sel_end = sel_line = 0; /* Nothing selected */
 HiliteControl (acc_button, 255); /* Dim accept button */
 while((cp = (*add_item>()) != 0) /* Call user for items */
 {
 il = (long)(cp); /* il = length of this item (chars) */
 max_wid = ((sw = (short)StringWidth (cp++)) >
max_wid) ? sw : max_wid;
 TEInsert (cp, il, hTE); /* Insert item in box */
 TEKey ('r', hTE); /* Start a new line */
 }

/*
 * Check if we need h-scroller. If so, activate it.
 * If active, set its range per longest string. Adjust
 * range in units of page scroll amount.

```

```

*/
if(!hce && max_wid > (view_rect.right - view_rect.left - 3))
{
 hce = TRUE; /* Turn on the scroller if needed */
 HiliteControl (h_scroll, 0);
}
if(hce) /* If it's on, range it to longest line */
 SetCtlMax (h_scroll, max_wid - (view_rect.right -
view_rect.left));

/*
 * Same for the vertical scroller. Activate it if number
 * of lines is greater than will fit on the screen. Always
 * set controlMax to number of lines x line height less
 * lines visible.
 */
if((*hTE)->nLines > lines_vis && !vce)
{
 vce = TRUE; /* Turn on scroller if needed */
 HiliteControl (v_scroll, 0);
}
if(vce) /* If it's on, range it to # of lines */
 SetCtlMax (v_scroll, ((*hTE)->nLines) - lines_vis) *
line_height);
break;

/*
 * CLEAR pressed. Axe everything, delete all items from
 the list, turn off the scrollers
 * and accept button. Bash the TEREcord's view_rect
 back to the original to remove scroll
 * effects. Reset state vars.
 */
case CLEAR: /* Axe all data in selection box */
 sel_start = sel_end = sel_line = 0; /* No selection */
 TEDeactivate (hTE); /* Hide selection */
 TESetSelect (0, 32767, hTE); /* Select everything */
 TEDelete (hTE); /* Axe it */
 HiliteControl (acc_button, 255); /* Dim button */
 hce = vce = FALSE; /* Deactivate & reset scrollers */
 hs_offs = vs_offs = 0; /* Back to upper left corner */
 (*hTE)->destRect.topLeft.all = dest_rect.topLeft.all;
 (*hTE)->destRect.botRight.all = dest_rect.botRight.all;
 SetCtlValue (v_scroll, 0); /* Reset controls */
 SetCtlMax (v_scroll, 0);
 HiliteControl (v_scroll, 255);
 SetCtlValue (h_scroll, 0);
 SetCtlMax (h_scroll, 0);
 HiliteControl (h_scroll, 255);
 max_wid = 0; /* Reset longest line */
 break;

/*
 * Click in selector box. Highlight the item (line of text) in
 which the click occurred.
 * This turns out to be rather easy. The line index can be
 determined by calculating
 * the distance of the click from the top of the TEREc's
 destRect and dividing by the line
 * height. Then use the "lineStarts" array to get the
 starting and ending indexes for
 * the line and calling TESetSelect to highlight the
 selection. Now call TEActivate

```

```

* to show the highlighting. Finally, turn on the Accept
 button.
*/
case SELBOX: /* Click in the selector box */
 if((*hTE)->nLines == 0) /* (skip it if no lines) */
 break;

/* Calculate line index */
sel_line = (loc_pt.v - (*hTE)->destRect.top) / line_height;

/* Deselect if click below last line. */
if(sel_line >= (*hTE)->nLines) /* Beyond last line */
{
 TEDeactivate(hTE); /* Hide selection */
 TETestSelect(32767, 32767, hTE); /* Deselect */
 sel_start = sel_end = sel_line = 0; /* No select */
 HiliteControl(acc_button, 255); /* Dim button */
 break;
}

/* Highlight selected text & turn on the Accept button */
sel_start = (*hTE)->lineStarts[sel_line]; /* 1st char */
sel_end = (*hTE)->lineStarts[sel_line + 1];
/* Index of char following sel'd line */
TETestSelect(sel_start, sel_end, hTE); /* Set select */
TEActivate(hTE); /* Highlight selected line */
HiliteControl(acc_button, 0); /* Activate button */
break;

default:
}
} /* END OF MAIN DIALOG LOOP ... */
#TEDispose(hTE); /* Junk TextEdit stuff */
#DisposeDialog(dp); /* Close & free heap space */
}

/*
* USER_FILTER - Filter dialog events to handle scroller and
 select box
*
* This sets up for Consulair Mac C. Your compiler's linkage
 may differ.
*/
user_filt()
{
#asm
;
; Inputs:
; 4(SP) Address of word to fill in with item number
; 8(SP) Address of the event record
; 12(SP) Dialog (window) pointer
;
Link a6,#0 ; No local automatics
Movem.L d1-d2,-(sp) ; Save regs
Move.L 8(a6),d0 ; D0 -> Item number word
Move.L 12(a6),d1 ; D1 -> Event record
Move.L 16(a6),d2 ; D2 -> Window record
Jsr _filt ; Call C for dirty work
Movem.L (sp)+,d1-d2 ; Restore regs
Unlk a6 ; "standard" Pascal routine exit ...
Move.L (sp)+,a0 ; A0 -> return point
Addq.L #6,sp ; Pop args

```

```

Addq.L #6,sp ; (don't ask why ...)
Move.B d0,(sp) ; Copy C result to Pascal return
Jmp (a0) ; Return
#endasm
}

/*
* This is the "real" filter function
*
* We must handle activating & updating of the scroller and the
 box area, but pass FALSE back
* so dialog manager does his thing on the other items. We
 must handle mouse downs in the
* scroller and text box, passing back TRUE, and pass FALSE
 back for everything else.
*
* Inputs:
* P1 -> word to receive item code
* P2 -> Event record
* P3 -> WindowRecord for dialog
*/
_filt(ip, ep, wp)
short *ip;
EventRecord *ep;
WindowPtr wp;
{
 short part;
 ControlHandle ch;
 int scroll_up();
 int scroll_down();

 SetPort(wp); /* We shouldn't have to but ... */

 switch(ep->what) /* Dispatch on event type */
 {
 /*
 * MOUSE CLICK
 */
 case mouseDown:
 loc_pt.all = ep->where.all; /* Preserve event point */
 GlobalToLocal(&loc_pt); /* We need local coord */
 part = (short)FindControl(&loc_pt, wp, &ch); /* Where? */
 /*
 * Try for one of our scrollers (ignore other controls)
 */

 if(part != 0 && (ch == v_scroll || ch == h_scroll))
 {
 /*
 * CLICK IN ONE OF OUR SCROLL BARS
 */
 switch(part) /* Clicked in our scroller */
 {
 case inUpButton: /* Up: Track w/action proc */
 TrackControl(ch, &loc_pt, scroll_up);
 return(TRUE);
 case inDownButton: /* Down: Track w/action proc */
 TrackControl(ch, &loc_pt, scroll_down);
 return(TRUE);
 case inPageUp: /* PageUp: Jump per Mac Interface */
 page_scroll(part, ch, -1);

```

```

 return(TRUE);
 case inPageDown: /* PageDown: Do Mac interface */
 page_scroll(part, ch, 1);
 return(TRUE);
 case inThumb: /* Thumb: Track w/no action proc */
 TrackControl(ch, &loc_pt, 0);
 box_scroll(); /* Jump to new scroller setting */
 return(TRUE);
 default:
 return(FALSE); /* Shouldn't happen (ha) */
 }
}
else if(PtInRect(&loc_pt, &box_rect))
/* Click in our selector box? */
{
/*
* CLICK IN SELECTOR BOX. Handled by
 ModalDialog() caller, not here!
*/
ip = SELBOX; / Fill in item code */
return(TRUE); /* Return to caller for action */
}
else
/*
* CLICK SOMEWHERE ELSE - LET MODAL-DIALOG
 DO IT
*/
return(FALSE);

/*
* UPDATE - Notice no BeginUpdate() & EndUpdate()? Why
 not? You figure that one out.
*/
case updateEvt:
 SetPort(wp); /* (really needed?) */
 DrawControls(wp); /* Crude, double-draws buttons */
 PenNormal(); /* Too bad if user needs pen */
 FrameRect(&box_rect); /* Draw box */
 TEUpdate(&view_rect, hTE); /* Draw text */
 return(FALSE); /* Let dialog manager do the rest */

/*
* ACTIVATE/DEACTIVATE - Is this needed? (yes, but
 why?)
*/
case activateEvt:
 SetPort(wp); /* (really needed?) */
 if(ep->modifiers & 1) /* Activate */
 {
 ShowControl(v_scroll); /* Draw scrollers */
 ShowControl(h_scroll);
 }
 else
 {
 HideControl(v_scroll); /* Erase scrollers */
 HideControl(h_scroll);
 }
 return(FALSE); /* Let dialog manager do the rest */

/*
* EVENTS WE DON'T TRAP
*/

```

```

 default:
 return(FALSE); /* Dialog manager does it */
 }
}

/*
* LOCAL UTILITIES
*/

/*
* PAGE_SCROLL() - Scroll a page per indicator
*
* Inputs:
* part Part code where first clicked down
* ch control handle
* dir direction (-1 = up, +1 = down)
*
* Outputs:
* none
*/

static page_scroll(part, ch, dir)
short part; /* Part where mouse first clicked */
ControlHandle ch; /* ->-> Control where mouse first clicked */
short dir; /* Direction code (see above) */
{
 Point cur_pt; /* Where is mouse now? */
 short amount;

/*
* First, calculate the "page size" in pixels. For V scroll, it is
 the viewRect height less the
* line height. For H scroll, it is half the width of the viewRect.
*/

 if(ch == v_scroll)
 amount = view_rect.bottom - view_rect.top - line_height;
 else
 amount = half_wid;

 amount *= dir; /* Change sign per requirements */

/*
* Now we must locally handle the mouse per the Macintosh
 Interface
* Guidelines. Look closely at this code and note what it
 really does ...
*/
 do /* Do this once even if mouse is */
 { /* up by now ... */
 GetMouse(&cur_pt); /* Get current mouse location */
 if((short)TestControl(ch, &cur_pt) != part)
 /* If out of original part, then */
 continue; /* don't do anything */
 #SetCtlValue(ch, GetCtlValue(ch) + amount); /* Page */
 box_scroll(); /* Page the text */
 } while(StillDown()); /* Keep it up till mouse released */
}

/*
* SCROLL_UP() - Scroll up the selector box
*/

```

- \* This routine is called back from the toolbox with (naturally) Pascal-flavored arguments
- \* on the stack. This routine should work with any compiler supporting inline assembler.
- \* If yours doesn't, just code the routine in "real" assembler & link it in. It's not compiler-
- \* dependent. Note the limits on `_SetCtlValue()` to prevent "rattling against the stops".

\*/

**static scroll\_up()**

```
{
#asm
;
; Inputs:
; 4(sp) Part code (int)
; 6(sp) Control handle (address)
;
controlMin EQU 20

Link a6,#0
Move.W 8(a6),d0 ; D0 = part code (W)
Beq @1 ; (0 means out of part region)
Move.L 10(a6),-(sp) ; Push control handle (for later)
Clr.W -(sp) ; Gets control value
Move.L 10(a6),-(sp) ; Push control handle
_GetCtlValue ; (SP) = current control value
jsr get_lh ; D0 = line height
Sub.W d0,(sp) ; (SP) = new control value
Move.W (sp),d0 ; D0 = new value
Move.L 2(sp),a0 ; A0 = Control Handle
Move.L (a0),a0 ; A0 -> scroller record
Cmp.W controlMin(a0),d0 ; Compare with controlMin
Bge @0 ; (in range)
Move.W controlMin(a0),(sp) ; Limit it to min
@0: _SetCtlValue
Jsr box_scroll ; call C text scroller
@1:
Unlk a6 ; "standard" Pascal routine exit
Move.l (sp)+,a0
Addq.l #6,sp
Jmp (a0)
#endasm
}
```

/\*

- \* SCROLL\_DOWN() - Scroll down the selector box

\*

- \* This routine is called back from the toolbox with (naturally) Pascal-flavored arguments
- \* on the stack. This routine should work with any compiler supporting inline assembler.
- \* If yours doesn't, just code the routine in "real" assembler & link it in. It's not compiler-
- \* dependent.

\*/

**static scroll\_down()**

```
{
#asm
;
; Inputs:
; 4(sp) Part code (int)
; 6(sp) Control handle (address)
```

```
;
controlMax EQU 22
```

```
Link a6,#0 ; no local automatics
Move.W 8(a6),d0 ; D0 = part code (W)
Beq @1 ; (0 means out of part region)
Move.L 10(a6),-(sp) ; Push control handle (for later)
Clr.W -(sp) ; Gets control value
Move.L 10(a6),-(sp) ; Push control handle
_GetCtlValue ; (SP) = current value
Jsr get_lh ; D0 = text line height (**SICKO**)
Add.W d0,(sp) ; (SP) = new control value
Move.W (sp),d0 ; D0 = new value
Move.L 2(sp),a0 ; A0 = Control Handle
Move.L (a0),a0 ; A0 -> scroller record
Cmp.W controlMax(a0),d0 ; Compare with controlMax
Ble @0 ; (in range)
Move.w controlMax(a0),(sp) ; Limit it to max
@0: _SetCtlValue
Jsr box_scroll ; call C text scroller
@1:
Unlk a6 ; "standard" Pascal routine exit ...
Move.l (sp)+,a0
Addq.l #6,sp
Jmp (a0)
#endasm
}
```

/\*

- \* BOX\_SCROLL() - Scroll text in selector box
- \* Changes TEREc's destRect per current scroller values. The scroller's value is equal to the
- \* number of pixels that the dest\_rect has been offset upward relative to the view\_rect.
- \* If both scrollers are at 0, you see the upper left of the destRect.

\*/

**static box\_scroll()**

```
{
short int dh, dv;

dh = hs_offs - (short)GetCtlValue(h_scroll);
/* Get scroll changes for X & Y */
dv = vs_offs - (short)GetCtlValue(v_scroll);

TEScroll(dh, dv, hTE); /* Scroll the destRect */
hs_offs -= dh; /* Update the offset values */
vs_offs -= dv;
}
```

/\*

- \* This is just a cheap way to get at this C global from assembly language without knowing its
- \* R5 offset as generated by the compiler's static access mechanism.

\*/

```
get_lh() /* Get text line height */
{
return(line_height);
}
```



# Engineer's Look at C

## MacPaint File Formats in C



Keith McGregor  
MacTutor Vol. 1 No. 7

The MacPaint file format is rapidly becoming the standard for transferring graphic information from one application to another. This article will explain that format and show you how to transfer your own images into MacPaint files.

### A Little Bit of QuickDraw

The Macintosh's screen is a bit image. That is, what appears on the screen is actually a collection of consecutive bits in memory that the hardware in the Mac interprets as screen dots (or pixels). If the value of one of these bits is 1, then the pixel corresponding to it will be black. If the bit is 0, then the pixel will be white.

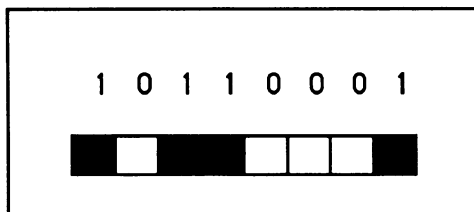


Figure #1. Bits represent pixels

The Mac screen is 512 pixels wide by 342 pixels tall. The bit image in memory takes up 175,104 bits (or 21,888 bytes with 8 bits per byte). We can consider the screen bit image as being 342 rows with 64 bytes per row. Due to the 68000 microprocessor in the Macintosh, every bit image must have an even number of bytes per row, and the rows must begin and end on word boundaries.

In C, the easiest way to define a bit image is with an array of characters. For example, if we wanted to define the Macintosh screen's bit image, the definition would be:

```
char MACSCREEN[342][64];
```

Figure 2: C Definition of a Bit Image

Bit images are manipulated by Quick-Draw through the use of bitmaps. A bit-map is a structure which points to a bit image and associates a coordinate system with it. The definition of a bitmap is:

```
struct ABitMap {
 char *baseAddr;
 short rowBytes;
 Rect bounds;
};
```

Figure 3: C Definition of a Bitmap

The baseAddr field points to the first byte (character) in the bit image. The rowBytes field contains the number of bytes per row in the bit image. The rectangle bounds provides the coordinate system for the bits. The top left corner of the rectangle is aligned with the first bit of the bit image.

Since we can treat bit images in C as arrays of characters, we don't always have to display them immediately. You can easily create a picture in a bit image off-screen, and later in your program "stamp" it onto the Mac's screen.

The Macintosh's ROM contains several QuickDraw calls that allow you to manipulate bitmaps (on- or off-screen). We'll look at two of these later in the program.

MacPaint images are kept in the data fork of a MacPaint file. The file is essentially a 512 byte header followed by a bit image that is 576 pixels wide by 720 pixels tall (or 72 bytes by 720 rows).

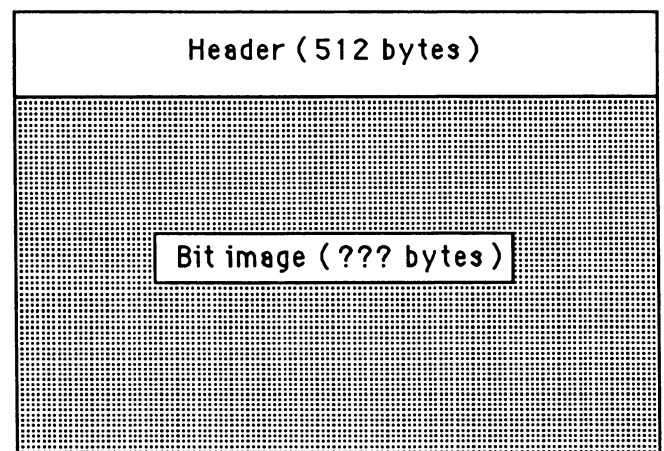


Figure 4: The MacPaint File Format

The 512 bytes of header contain version information, pattern definitions, and an unused area for future expansion. If

you change the contents of this section with care, you can set up your own patterns. If the version number is zero, the default patterns are used. If the number is not zero, then the patterns in the header are used.

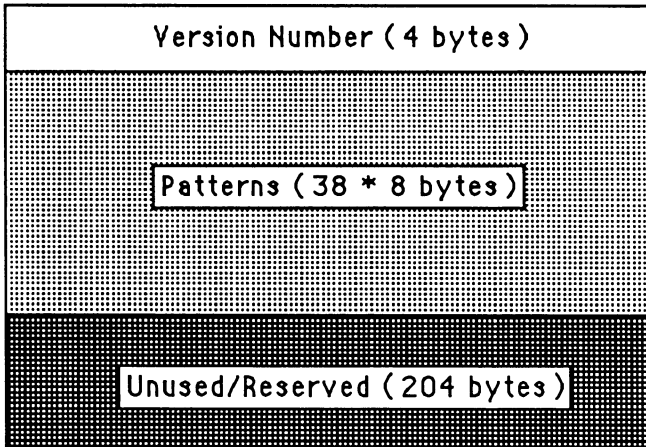


**Fig 5: The Paint File Header Format**

The MacPaint file bit image is considerably larger than the Macintosh screen's bit image. In fact, if you were to store a bit image 72 bytes by 720 rows directly onto disk, each MacPaint file would take a minimum of 52K bytes! To make these files smaller, each row of pixels has been compressed using the PackBits routine in the Macintosh ROM. Using PackBits, the typical MacPaint file compresses down to around 10K bytes.

### Dipping in the Bit Bucket

To make your program transfer an image into a MacPaint file, you need only to copy the bitmap containing the image into the compressed MacPaint file format. To be able to do this, you need to use two ROM routines, CopyBits and PackBits.

CopyBits transfers the contents of one bitmap into another. If you want, you can specify a mask region in the destination bitmap to prevent CopyBits from destroying the entire image. CopyBits will shrink or expand the contents of the source bitmap to fit the destination bitmap's rectangle. To call CopyBits from C, use this format:

```
CopyBits(&srcMap, &dstMap,
 &srcRect, &dstRect,
 copymode, &maskRegion);
```

**Figure 6: Calling CopyBits from C**

where "copymode" is one of the source transfer modes:

|         |            |
|---------|------------|
| srcCopy | notSrcCopy |
| srcXor  | notSrcXor  |
| srcOr   | notSrcOr   |
| srcBic  | notSrcBic  |

**Figure 7: Source Transfer Modes**

PackBits compacts a string of bytes by compressing runs of equal bytes. You call PackBits from C like so:

```
PackBits (&srcPtr, &dstPtr,
 numbytes);
```

**Figure 8: Calling PackBits from C**

where "numbytes" is the number of bytes pointed to by "srcPtr."

Initially, "dstPtr" should point to the first destination byte. The PackBits routine will move "dstPtr" to point to the next available byte. To find out how much space the original bytes were compacted into, you need to subtract the original location of "dstPtr" from its current location.

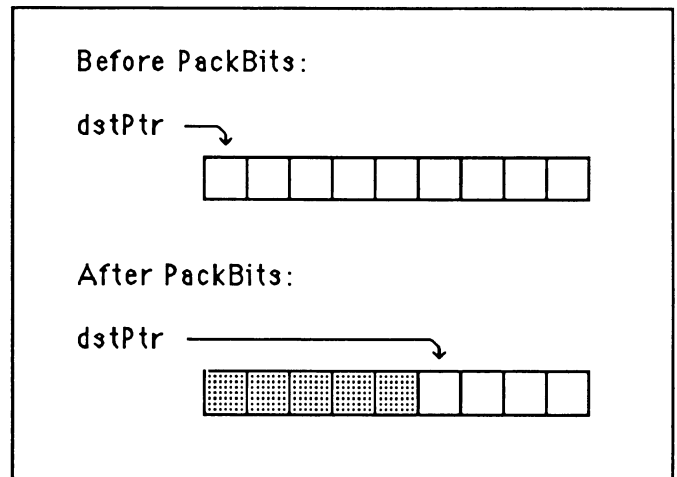


**Fig 9: How PackBits Moves "dstPtr"**

### About the Program

This program will demonstrate how to "cut" a part of an image into a MacPaint file. It is written for the Consulair Mac C system and Toolkit. You'll probably need to change things a bit to make it work with other versions of C.

The program opens a single window and draws a few inverted rectangles in it. Then, a gray flashing rectangle will appear and follow the position of the cursor. Use the mouse to position this rectangle where you want, and then press the mouse button. The image inside the rectangle will be cut from the screen (using CopyBits) and transferred into a

MacPaint file called "DummyFileName" (using repeated calls to PackBits).

The key routine, MakePaintFile, takes as input a C string containing a filename, and a pointer to a bitmap of arbitrary size. It then creates that file (if possible) on the active disk drive, sets the file signature and type to a MacPaint file type, and compresses that bitmap into the file using PackBits. If the bitmap is smaller than 576 by 720 (as will usually be the case), the bitmap is padded with white space. If the bitmap is larger than 576 by 720, the bitmap is clipped to 576 by 720.

```
/*
 * MakePaint.h
 * a C routine to create a MacPaint file
 * from a given bitmap
 *
 * (c) 1985 by Keith McGreggor for MacTutor
 */
#define WRITEONLY 7

MakePaintFile(myfilename, mybitmap)
char myfilename[63];
struct ABitMap *mybitmap;
{
 char dstbuf[511],srcbuf[511];
 char *srcPtr;
 char *dstPtr;
 char *mypointer;
 short i,j,dstBytes;
 FILE myfile;
 short t,b,vsize,hsize;

 // attempt to create the file
 myfile = creat(myfilename, WRITEONLY);
 if (myfile != 0) {
 // turn the file into a MacPaint file
 // and write out a 512 byte header
 // full of zeros (we're not using any of
 // our own patterns)
 SetFileSignature(myfilename,'MPNT');
 SetFileType(myfilename,'PNTG');
 for (i = 0; i < 512; i++) fputc(0,myfile);
 // figure out how big the bitmap is and set up
 // a general pointer to the bit image
 mypointer = mybitmap->baseAddr;
 hsize = mybitmap->rowBytes;
 vsize = (mybitmap->bounds.bottom)
 - (mybitmap->bounds.top);
 // now, write out 720 rows of bytes
 for (j = 1; j <= 720; j++) {
 // reinitialize source and destination pointers
 srcPtr = &srcbuf[0];
 dstPtr = &dstbuf[0];
 // copy the next row of bytes into srcbuf[],
 // clipping or expanding where necessary
 for (i = 0; i < 72; i++) {
 if ((i < hsize) && (j <= vsize)) {
 srcbuf[i] = *mypointer;
 mypointer++;
 }
 }
 }
 }
}
```

```
 }
 else srcbuf[i] = 0; // 8 white bits
 }

 // compress srcbuf[] into dstbuf[]
 #PackBits(&srcPtr,&dstPtr,72);
 // figure out how much compression occurred
 // and write those bytes out to the file
 dstBytes= (short)(dstPtr-&dstbuf[0]);
 for (i = 0; i < dstBytes; i++)
 fputc(dstbuf[i],myfile);
 }

 // all 720 lines have been written, so
 // close up everything and return
 close(myfile);
}
}
```

---

```
/* CutToPaint.C
 * A sample program to illustrate
 * the use of PackBits and CopyBits
 *
 * (c) 1985 by Keith McGreggor for MacTutor
 */
#include <Stdio.h>
#include <MacCdefs.h>
#include <Window.h>
#define TRUE 0xFF
#define BUTTONNOTPRESSED !#Button()
#define GRAY &(QD->gray)

// allocate the application's window

WindowPtr mywindow;
Rect mywindowrect = { 40, 5, 300, 507 };

// allocate the tracking variables

Rect trackingrect{0,0,100,100};
short globx,globy,incx,incy;

// allocate a bitmap and bit image to
// hold the cut bits

struct ABitMap{
 char *baseAddr;
 short rowbytes;
 Rect bounds;
};

struct ABitMap targetmap;
char theactualbits[14][100];

#include "MakePaint.h"
/* ----- */
// draw some inverted rectangles
// just to have something to "cut"

DrawSomeStuff()
```

```

{
Rect temprect;
short i;

#MoveTo(10,240);
#DrawString("\020Press the button!");
for (i=0;i<200;i += 10) {
 #SetRect(&temprect,i+20,i+20,i+100,i+40);
 #InvertRect(&temprect);
}
}

```

```

/*-----*/
// Open a window for drawing
// and do general housekeeping

```

#### InitializeOurSystem()

```

{
#InitDialogs(0);
mywindow = (WindowPtr)#NewWindow(0,
 &mywindowrect,"021A Window for bits",
 TRUE,0,-1,TRUE,0);
#SetPort(mywindow);
DrawSomeStuff();
#FlushEvents(-1,0);
#InitCursor();
#PenSize(2,2); // set up pen for
#PenMode(patXor); // drawing the
#PenPat(GRAY); // ghost rectangle
}

```

```

/*-----*/
// Draw the ghost rectangle

```

#### DrawCurrentPosition()

```

{
#FrameRect(&trackingrect);
}

```

```

/*-----*/
// Find out where the mouse is and
// update global variables

```

#### GetNewMouse()

```

{
Point mypoint;

#GetMouse(&mypoint);
#GlobalToLocal(&mypoint);
incx = mypoint.h-globx;
incy = mypoint.v-globy;
globx = mypoint.h;
globy = mypoint.v;
}

```

```

/*-----*/
// move the ghost rectangle to track
// the motion of the mouse

```

#### UpdatePostion()

```

{
#OffsetRect(&trackingrect,incx,incy);
}

```

```

/*-----*/
// First, move around a ghost rectangle
// until the button is pressed.
// Then, create a bitmap and copy
// whatever is inside the rectangle to
// the bitmap.

```

#### CutARectangle()

```

{
globx = 0;
globy = 0;
while (BUTTONNOTPRESSED) {
 DrawCurrentPosition();
 GetNewMouse();
 DrawCurrentPosition();
 UpdatePosition();
}
#FlushEvents(-1,0);
#LocalToGlobal(&trackingrect.topleft);
#LocalToGlobal(&trackingrect.botRight);
targetmap.baseAddr = &theactualbits[0][0];
targetmap.rowBytes = 14;
targetmap.bounds.left = 0;
targetmap.bounds.top = 0;
targetmap.bounds.right = trackingrect.right-trackingrect.left;
targetmap.bounds.bottom = trackingrect.bottom-
 trackingrect.top;

```

// mywindow -> portBits is the Macintosh screen bit image

```

#CopyBits(&(mywindow->portBits), &targetmap,
 &trackingrect,
 &(targetmap.bounds),
 srcCopy,
 0);
}

```

```

/*-----*/
// call makepaintfile to save the
// bitmap away as a macpaint file
// named "dummyfilename"
// (note: see MakePaint.h listing)

```

#### SaveItAway()

```

{
MakePaintFile("DummyFileName",&targetmap);
}
/*-----*/
// Our main program

```

#### main()

```

{
 InitializeOurSystem();
 CutARectangle();
 SaveItAway();
 #ExitToShell();
}

```





# C Workshop

## Function Resources



Robert B. Denny  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 8

Several weeks ago, a group of us were having a "programmer's lament" discussion, centering around some of the seemingly needless holes in the Mac's development environment, in particular the Macintosh Development System (MDS) assembler/ linker package. I happen to know some of the private history behind it's implementation, and why it lacks librarian and selective linking support.

It is alleged that the developer pleaded for those features but "Apple" refused to pay for them, citing lack of need. Ah, well, such is the plight of us old-timers who got spoiled 5 years ago on 16-bit systems which had 32K of memory!

The conversation meandered to more fertile ground. We discussed such things as sharing code between applications run- ning under Switcher, a common error alert system, and how one could imple- ment a "package-like" resource that could be loaded, locked and jumped into at run time ... without linking it in ... written in C (or at least most of it).

### Function Resources

The result of that discussion is a way to implement what I'll call *Function Resources*. A function resource (FR) is simply a resource that you can read in, lock down and call as a C function. It is not linked into your program (as would be a segment).

### Caveat

The techniques presented in this article are specific to the Apple MDS assembler/linker and to the Consulair Mac C compiler, which uses the Apple MDS system for assembly and linking. If you are using another development environment, the ideas presented here will still be of use, within the limits imposed by your development system.

I wrote my first function resource in assembler to get a feel for the way the MDS linker handles code that is assembled following a RESOURCE directive in the assembly source. That FR consists of a list of English language error messages for all of the system error codes, including those generated by the AppleTalk network drivers, and an alert display function.

The purpose was to provide a C-callable "package" that would put up a meaningful alert for any system error, and return an indication of whether the user pressed a "Quit" button or a "Resume" button. The resource contains all of the error messages, not the application. It isn't exactly in the spirit of the Mac interface, but it's very handy.

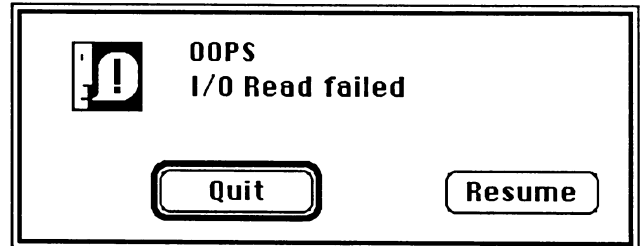


Fig. 1 Status Code in Tables

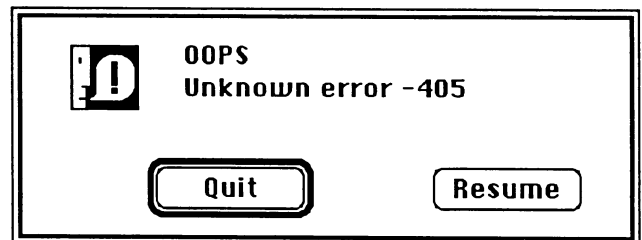


Fig. 2 Status Codes not in Tables

### Techniques

The application program contains a very small *transfer function* which simply loads the FR resource, locks it down, then does a JSR to it's beginning. Upon return from the FR, the transfer function unlocks the resource, which has the "purgeable" attribute, then returns the FR's function result to the caller.

Since the FR is not linked with the application, any sharing of data must be via parameters supplied with the call. However, there is nothing to prevent you from defining a "work area" in your program, then passing a pointer to the work area as a parameter to the FR.

If you write the FR itself so that it contains no read-write static data, then a single copy of the FR can be called by multiple applications and/or desk accessories. Such an FR is said to be *re-entrant*. R/W static data is called *impure* data, while read-only static data is called *pure* data. For a routine to be re-entrant, it must not contain impure data. FR's can have read-write data if it is allocated on the stack at entry, like automatic variables in C.

With this as a background, lets cover the steps needed to create and use a Function Resource:

1. Write the FR in assembler or C. Begin it with the assembler's RESOURCE directive.

2. Assemble and link the FR. Use the linker's /RESOURCES option prior to the FR module name to create a resource file.
3. Optionally, combine the FR with other resources such as dialogs, alerts and window templates via an RMAKER run.
4. Write a transfer function which will load and lock the resource and transfer control to the code contained therein.
5. Link the transfer function into your program.
6. Call the transfer function to use the function resource.

### Creating the Function Resource

The MDS assembler and linker provide the support needed to create the function as a resource without any additional hacking with utilities such as "Fedit" or "ResEdit". The assembly must start with a RESOURCE directive, declaring the remainder of the module as resources rather than normal code. For example:

```
RESOURCE 'PROC' 2000 'Foo 1.2' 32
```

This declares the module as a resource, with resource ID of 2000, resource type of "PROC" and with the "purgeable" attribute bit set. The type of PROC is an arbitrary choice on my part. For an FR written in Mac C, put the RESOURCE directive inside #asm/#endasm at the top of the source file. The optional resource name can be anything.

The MDS linker is smart enough to properly handle relocatable references in assembled resources. On the other hand, it cannot resolve inter-module references when linking resources. This means that your FR must be written to assemble as a single .REL file. You can have several source files, but they must be pulled together at assembly or compile time by text "include" directives. Given that you have your FR's .REL file, use the following linker commands to create the resource file, ready to use:

```
/OUTPUT MyFR.PROC
/Globals -0
/Type 'RSRC'
/Resources
MyFR

$
```

The /OUTPUT directive gives the resource file its name. The suffix ".PROC" is an arbitrary choice on my part, the name may be anything reasonable. The reason for the /GLOBALS directive will be covered later, just be sure it's there and you'll be safe. The "-0" is required because the linker wants a negative value for the parameter. The file type "RSRC"

indicates the file contains generic resources. The /RESOURCES directive indicates the beginning of resource data.

The result of assembling and linking as just described is a resource file which contains a purgeable resource of type PROC, with resource ID of 2000, which consists of code beginning at the first location in the resource.

### A Simple Function Resource

The following is the MDS assembly code for the error alert FR. The tables are not complete but it should be clear how to complete them.

Mac C uses registers D0-D6 to pass arguments, returning the function result in either D0 or A0, depending on whether the function returns a scalar or a pointer, respectively. Remember that the FR is called via the transfer function and not directly from the C program.

The text displayed in the alert box is formatted using the Dialog Manager's ParamText() function. If the error code and its English message are in the tables, the display is in English. If not, the toolbox "NumToStr" binary to decimal ASCII conversion "package" is called to convert the error code and it is displayed as "unknown error N". See the RMAKER control file which follows the assembly source for more on the use of ParamText().

Figure 1 shows what the error alert looks like for a known status code, Figure 2 shows its appearance for an unknown status code. The "user prefix" is the word "OOPS" followed by a carriage return ('\r').

```
;
; ** PERROR **
;
; Function resource for displaying system/AppleTalk
; error alert.
;
Include MacTraps.D
Include SysErr.txt ; System error codes
Include AtalkEqu.D ; AppleTalk error codes
;
RESOURCE 'PROC' 2000 'Perror 1.0' 32
;
; P-string format by default
STRING_FORMAT 3
;
; ENTRY:
; D0.W = Error/status code
; D1.L -> P-String to precede error message text
; D2.W = Resource ID of alert box to use (normally
; 2000)
;
; BASE:
Link a6,#-32 ; Workspace for NumToStr
Move.W d2,d7 ; Save Alert ID across ParamText
Move.L d1,-(sp) ; -> Caller's message for
; ParamText
Lea codes,a0 ; A0 -> base of code table
@10:
```

```

Move.W (a0)+,d3 ; D3 = table code (a0 -> offset)
Beq.S @20 ; (oops, end of table)
Cmp.W d0,d3 ; Matched?
Beq.S @30 ; (yes, display it)
Addq #2,a0 ; A0 -> next code
Bra.S @10

@20:
; Code not in table. Display "unknown error nn"
Move.L a6,a0 ; A0 -> temp buffer
Ext.L d0 ; Sign-extend error code in D0
Clr.W -(sp) ; Selector for NumToStr
_Pack7 ; A0 -> P-string of error code
Pea 'Unknown error ' ; P2 = Our prefix
; (for ParamText)
Move.L a0,-(sp) ; P3 = Error code string
; (for ParamText)

Bra.S @40
; Code found in table, we have English message
@30:
Clr.L d0 ; Zero out D0.L
Move.W (a0),d0 ; D0 - offset to string
Lea Strings,a0 ; A0 -> base of strings
Add.L d0,a0 ; A0 -> our string
Move.L a0,-(sp) ; P2 = Error message
; (for ParamText)
Pea L999 ; P3 = nothing (for ParamText)
; Common code to display the alert
@40:
Pea L999 ; P4 = nothing (for ParamText)
_ParamText ; Set the text (wipes d2!)
Clr.W -(sp) ; For function result
Move.W d7,-(sp) ; P1 = alert ID
Clr.L -(sp) ; Nil ProcPtr
_StopAlert ; Put up the alert
Move.W (sp)+,d0 ; Return alert function result
Unlk a6 ; Clean up
Rts ; Return to application's transfer
; function

;
; The following table contains ordered pairs consisting of ; a
; system error code followed by the byte offset into a
; list of strings of the error message for that error code.
;
CODES:
dc.w controlErr, 0
dc.w statusErr, (L2 - STRINGS)
dc.w readErr, (L3 - STRINGS)
dc.w writErr, (L4 - STRINGS)
dc.w badUnitErr, (L5 - STRINGS)
dc.w unitEmptyErr, (L6 - STRINGS)
dc.w openErr, (L7 - STRINGS)
dc.w closErr, (L8 - STRINGS)
dc.w dRemovErr, (L9 - STRINGS)
dc.w dInstErr, (L10 - STRINGS)
dc.w abortErr, (L11 - STRINGS)
dc.w notOpenErr, (L12 - STRINGS)
additional codes & message offsets here
dc.w 0 ; End of table
;
; This table contains the error messages, addressed by
; the offsets contained in the previous table.
;

```

```

STRINGS:
L1: dc.b 'I/O Control failed'
L2: dc.b 'I/O Status failed'
L3: dc.b 'I/O Read failed'
L4: dc.b 'I/O Write failed'
L5: dc.b 'Bad unit number'
L6: dc.b 'Unit is empty'
L7: dc.b 'I/O Open failed'
L8: dc.b 'I/O Close failed'
L9: dc.b 'Cannot remove open driver'
L10: dc.b 'Driver not found'
L11: dc.b 'I/O aborted by KillIO'
L12: dc.b 'I/O to unopened driver'
additional messages here
L999: dc.b 0,0 ; Addressable empty string
END

```

Assuming the above was assembled to a file called `PERROR.REL`, the next step is to link the REL file into a 0-based image in resource format. The linker control file is shown below:

```

/OUTPUT Perror.PROC
/Globals -0
/Resources
Perror

$

```

The RMAKER control file for the error dialog FR combines the PROC resource with an alert box and item list. Note the `statText` item with the meta-characters of the form "`^n`" are used with the Dialog Manager `ParamText()` call to set up the alert for display.

```

Perror
RSRC????
* Merge in the function resource
Include Perror.PROC

```

```

Type ALRT
,2000
40 96 138 416
2000
5555

```

```

Type DITL
,2000
3

```

```

Button
68 60 88 130
Quit

```

```

Button
68 190 88 260
Resume

```

```

StaticText Disabled
7 72 64 310
^0^1^2^3

```

## The Transfer Function

The transfer function is linked with the application. It provides the C-callable generic service needed to load the resource, lock it down, call it, unlock the resource and return. The following transfer function is written to be "generic", that is, independent of any particular FR, so that it may be used to call various FR's from Mac C. The first parameter to the transfer function is the resource ID of the PROC resource containing the desired FR. For the error alert FR, this is 2000. The remaining parameters are passed directly to the FR after removing the first parameter (the PROC ID). You supply the code to handle the case where the PROC resource can't be loaded.

```
; Inputs:
; D0.W = Resource ID of PROC
; Other D-regs contain args for proc
; Outputs:
; Returns FR's D0
;
; WARNING: Supports a maximum of 7 parameters
; following PROC ID.
;
; Include MacTraps.D
; XDEF DoProc
DoProc:
 Movem.L a1-a5,-(sp) ; Just in case ...
 Clr.L -(sp) ; Gets handle to FR
 Move.L #'PROC',-(sp) ; Resource type of FR
 Move.W d0,-(sp) ; Resource ID
 _GetResource ; Load FR resource
 Move.L (sp)+,d0 ; D0 ->-> FR? (need test)
 Beq @10 ; (didn't get it!)

 Move.L d0,a0 ; A0 ->-> FR
 MoveM.L d1-d6,-(sp) ; Save d-parameters
 MoveM.L (sp)+,d0-d5 ; Restore parameters
 ; shifted down in reg's
 Move.L 4(sp),d6 ; Get next parameter
 Move.L a0,-(sp) ; Save handle to FR
 Bset.B #7,(a0) ; Lock it down
 Move.L (a0),a0 ; A0 -> FR's entry
 Jsr (a0) ; Call perror
 Move.L (sp)+,a0 ; A0 ->-> FR
 Bclr.B #7,(a0) ; Unlock the FR
 MoveM.L (sp)+,a1-a5 ; Restore a-regs
 Rts ; Return FR's D0
@10:
 Handle resource load error here
 Rts
END
```

## Using the "Perror" Function Resource

To call the error alert FR from a Mac C program, issue a function call of the following form:

```
DoProc(2000, stat, prefix, 2000);
```

where "stat" is the 16-bit system status code for which to display the alert, "prefix" is a P-string containing text to be displayed in the alert prior to the English language error message, and the last 2000 is the resource number of the alert to use for the display. See the RMAKER file above for the description of the alert-2000 box used.

One convenient method of handling errors in Mac C programs is to use the "signal" mechanism to break out of the normal control flow and go to an error handler that invokes the Perror alert FR to display the error in an alert. For example:

```
if(stat = CatchSignal())
{
 DoProc(2000,stat,"005OOPS\r",2000);
 Whatever other recovery & cleanup
}
```

## Writing Function Resources in C

The "Perror" function resource is a simple example, written in assembly language to make the concept clear. Typically, however, you'll want to implement function resources in C, with a minimum of assembly "glue".

This is straightforward if you need only have automatic variables. Things get stickier if you want to have multiple functions in the FR, with data having module-wide scope. An additional complication arises if you wish to access the calling application's QuickDraw variables (to set patterns, for example).

Mac C normally uses register A5 as a base register for accessing application globals. In fact, this is a Macintosh programming convention; the toolbox expects A5 to point to the boundary between application parameters and application globals.

Static data is declared by the compiler using assembler "DS" directives. The linker collects these and "assigns" storage by computing a negative offset for use with a base register for each global item. Normally, this base register is A5. For example:

```
static int foo;
...
foo = func();

expands to (approximately):

FOO: DS.L 1
 JSR FUNC
 MOVE.L D0,FOO(A5)
```

Most toolbox calls require A5 to point to the "magic place". Therefore, an FR written in C should use a base register other than A5, usually A4. Fortunately, the Mac C compiler has an option to specify what A-register it is to use as the static data base register. This feature was meant for use by desk accessories written in C:

```
#Options R=4 /* Use A4 for statics */
```

Now for the final touch. If we are to keep the FR re-entrant, then it cannot contain read/write static data. But suppose we want to have read/write variables with module-wide scope? Here's how.

Declare the "static" variables as usual. Then reserve a chunk of space on the stack as a big automatic array belonging to the "first" FR function, the one which is called by the transfer function. Then, immediately on entry to the FR, bash A4 to point to the last cell in the automatic array. Thus, the automatic array serves as the "globals" area for the FR, based on A4, and the various functions in the FR may access the variables as if they were statically declared in the outermost (module) scope. And the FR is still re-entrant because the space is allocated at run-time on the caller's stack.

When the linker computes the negative offsets to assign to the static variables, it assumes that A5 is being used for the base register and that the statics are the application's globals. This being the case, it automatically reserves space for the QuickDraw globals. This causes the "first" static to have an offset of -200 hex.

This offset can be suppressed by including the linker directive "/GLOBALS -0" in the control file used to link the FR. That's "minus zero" ... needed because of a bug/feature in the linker. It wants to see a negative value with the /GLOBALS switch and won't accept zero. But minus zero is OK. The linker command file shown above contains this directive.

There is one VERY important caveat here. Static initializers will not work. Therefore, the FR must manually initialize the static data, Pascal style upon FR entry (after bashing A4, of course).

Perhaps you're saying to yourself, "Is all of this worth it?" Consider the uses for re-entrant run-time loadable functions.

"Packages" similar to the Standard File package can be implemented using this technique. The second FR I wrote uses the lookup dialog described last month to allow selection of named objects on an AppleTalk network.

A family of related applications could share functions contained in a single resource file. Remember that the code in the FR is not linked with the application, therefore reducing the disk space used by the program file.

Common code could be shared between applications running under the Switcher by loading the FR into the system heap. This would save both memory and disk space.

### A Template Function Resource in C

Next we'll look at a template for writing FR's in C. The techniques for linking and transfer are the same as those already presented. The DoProc() transfer function will work just fine with the C language FR

```
#asm
RESOURCE 'PROC' 1234 'Templ_1.0'
Include MACTRAPS.D
```

```
#endasm
```

```
/*
 * Declare pseudo-static variables here, then
 * define COM_SIZE equal to the total number
 * of bytes needed for the pseudo-statics.
 */
static int a;
static short b;
struct QDVars *QD; /* Our QD pointer */
#define COM_SIZE nn

/*
 * Entry is here, at the beginning of the module
 */
Func(p1, p2)
int p1; /* Declare these as needed */
int p2;
{
 char CommonVars[COM_SIZE]; /* Pseudo statics */
 Ptr appl_port; /* Saves caller's port */
 /*
 * Preliminary set-up stuff.
 */
 #GetPort(&appl_port); /* Save caller's port */
 LocInit(&(CommonVars[COM_SIZE-1]));
```

*The rest of the main function goes here*

```
#SetPort(appl_port);
return(result);
}
```

*Other functions may be coded here. References to the "static" variables are normal.*

```
#asm
;
; Initialization and A4 hacking routines
;
save_a4: dc.l 0 ; A4 pointer is kept here
grafSize equ $CE ; Size of QD's variables

LocInit:
 Lea save_a4,a0 ; Dumb 68000 designers!
 Move.L d0,(a0) ; Initialize our "statics" base
 Move.L d0,a4 ; A4 -> our pseudo-statics
 Move.L a5,d0 ; D0 -> QD's magic cell
 Sub.L #grafSize,d0 ; D0 -> QDVars
 Move.L d0,QD(a4) ; Save it in our common vars
 Rts

;
; This routine is used to re-establish our A4 context for
; functions which are called back from the toolbox, such
; as action routines for TrackControl(), and "filter procs"
; for ModalDialog().
;
GetA4:
 Move.L save_a4,A4 ; PC-relative, eh?
 Rts
```

Notice the convenience of PC-relative addressing such as that used for accessing "save\_a4" above. When I first started working with the 68000, I nearly lost my mind until I discovered that PC-relative is illegal as a destination addressing mode! My reference manual neglected to mention that little fact.

This wraps up our discussion of function resources. It is my hope that, despite the orientation to Apple's MDS and to Mac C, the information given here will be of use to C programmers in general. I'd be interested to hear from readers on how they implemented these ideas with their C systems.

### Update Event Handling Revisited

Last month, the C Workshop presented a large example source listing of a callable function that implements a general purpose "selector dialog" similar to that used by the Standard File package. Along with bringing together the ideas discussed over several previous issues, the example showed how to use the "filter procedure" hook provided by the ModalDialog() service of the Dialog Manager.

The update event handler in the filter procedure is incorrect. The "test program" I used to check out the selector dialog function did not uncover a glaring error. Please refer to last month's article for background information.

It turns out that if you do anything at all with update events in your filter procedure, you must handle all update activities. The update procedure must start with a call to BeginUpdate(), which copies the update region into the "vis" region and sets the update region to NIL.

I tried to be tricky and skip the call to BeginUpdate(), draw only the "special" items in the dialog box, then return FALSE to ModalDialog, signalling it that it should handle the

update event. Well, under the test program, it worked fine, but in a real application, I discovered that the lack of a call to BeginUpdate() caused my "special stuff" to get redrawn lots of unnecessary times.

So I put in the call to BeginUpdate(), figuring that if I didn't call EndUpdate() and I returned FALSE, ModalDialog() would still finish the job. I should have known...

When ModalDialog() gets control back from the filter procedure and it sees that the event is an update event, it immediately calls BeginUpdate(). This sets the visRgn to the update region, which was set to NIL by my call to BeginUpdate() in the filter procedure.

The moral of the story is that filter procedures which call BeginUpdate() must do all required updates to the dialog and then return TRUE to ModalDialog(). This indicates that the event was completely taken care of by the filter proc.

Rather than show the corrected code for the update portion of the filter procedure, I'll leave the bug fixes as an exercise. Some hints: Use TextBox() to draw statText items. A single call to DrawControls() will update all buttons and scrollers in the dialog. Call TEUpdate() for each EditText item in the dialog.

### Feedback Wanted

The articles presented in the C Workshop over the past few months have been highly technical in content. This is in keeping with our stated purpose of providing high quality technical information oriented toward developers. I would like to know if there are a significant number of readers for whom the article content has been too advanced. Until next month.



# C Workshop

## Using the Vertical Retrace Manager



Robert B. Denny  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 9

This month, we look at some obscure but very useful features of the Macintosh operating system, and combine them in a complete example program, a CRT saver. The CRT saver does not require a desk accessory slot, nor must it be run as an application to get it installed. The example program also illustrates techniques for using low-level operating system features from C. As usual, the information presented here is meant to supplement that in Inside Macintosh.

### The Vertical Retrace Manager

The Vertical Retrace Manager is used to schedule repetitive tasks at timed intervals. It gets its name from the fact that it is activated when the electron beam that paints the Mac screen "snaps back" to its starting place after painting the entire screen from top to bottom. This *vertical retrace* happens 60 times a second.

Making use of the Vertical Retrace Manager is easy. Simply fill in a data structure with a pointer to the task procedure you want to schedule and the time delay (in ticks). Then issue a `Vinstall()` with the pointer to that data structure. At each vertical retrace, the delay is decremented. When the number of ticks gets to zero, the task is called.

After the task completes, the Vertical Retrace Manager checks the number of ticks. If it is still zero, nothing further is done. If the task re-loads the tick count, the whole process is repeated. Thus, to have the task periodically scheduled, simply have it re-load the tick count with the desired interval.

The data structure is a *queue element*, a structure used for various purposes throughout the Mac operating system. The different flavors of queue elements have one thing in common. As suggested by the name, queue elements are used in applications where things are placed on a queue. Queues are typically used to serialize processing or to otherwise establish order. Generically, a queue element may be represented as:

```
struct QE
{
 struct QE *QLink; /* Link or NULL */
 short QType; /* Type of element */
 char QData[1]; /* 1st byte of rest of data */
};
#define QElem struct QE
```

The type of queue element is indicated by the value in the `QType` field. Types include `IOQType`, for input-output queues, `DrvType`, for the drive queue, `EVType`, for the event queue, `FSQType` for file system queues, and `VType` for the vertical retrace queue. The contents of the rest of the queue element is specific to the type. A structure definition for a vertical

retrace queue element is shown below:

```
struct VB
{
 struct VB *qLink; /* Queue link pointer */
 short qType; /* Always 1 (VType) */
 ProcPtr vblAddr; /* -> Task to be scheduled */
 short vblCount; /* Delay in ticks */
 short vblPhase; /* Phase (see below) */
};
#define VBLTask struct VB
```

When you call `Vinstall()`, the operating system places your queue element at the end of the vertical retrace queue. This means that your task will get activated following those already scheduled. The `VblPhase` field is used to interleave tasks which are repetitively scheduled with the same delay so that they are executed in separate "slots".

You might be wondering why "VBL" is used when describing things associated with vertical retrace activity. The VBL stands for "vertical blanking" a synonym for vertical retrace. The two are interchangeable.

There are a few things to be aware of when writing a VBL task. The task gets run *asynchronously*. Whenever the vertical retrace occurs, the current process is interrupted and control transfers to the Vertical Retrace Manager, which saves registers D0-D3 and A0-A3, then calls each task whose tick count has reached zero. When the last task completes (with an RTS), those registers are restored.

This has major consequences. First, the VBL task must save and restore any registers (other than D0-D3 and A0-A3) that it uses. Second, the VBL task must never make Memory Manager requests which allocate or free memory. Since many system services (e.g., resource manipulation) generate Memory Manager requests, this severely restricts activities inside a VBL task. Third, the execution time must be kept short because there may be many VBL tasks scheduled for a particular tick, and all must complete before the next retrace, 16.67 milliseconds later.

The most common way for an application to use a VBL task is to have it control flags and/or timers that are used by the application in its event loop. This way, the timing of application activity is controlled by the VBL task, while the time-consuming processing is done in the application itself.

Why no Memory Manager activity in a VBL task? Since the task is activated asynchronously, it may interrupt the application process right in the middle of Memory Manager services. At that time, the memory management data structures may be in an inconsistent state; a block may be "partially" deallocated, for example. Trying to do something

else at that time would cause the whole thing to become corrupt.

VBL tasks have many uses. Any time you have animation to do, consider using a VBL task to make the motion smooth. If you rely on the consistency in timing of your application's event loop, you may be disappointed. When your system gets AppleTalk installed, for example, there can be a lot of asynchronous activity, which will upset your timing loops. Have a VBL task set a flag indicating that a certain amount of "real" time has elapsed, and use this knowledge to animate. Remember that you get sixty frames a second on the Mac screen, and the VBL task can schedule things sixty times a second, so there is no loss in scheduling bandwidth.

The Mac system contains several "standard" VBL tasks which handle the following:

- Check whether the stack and heap are getting too close to each other. This is the "stack sniffer" (every tick).
- Increment the global variable Ticks, the number of ticks since system startup (every tick).
- Handle cursor movement (every tick).
- Deglitch the mouse button and post mouse events (every other tick).
- Post a disk-inserted event if a disk was inserted (every 30 ticks).

In the CRT saver, the VBL task periodically examines the amount of time that has elapsed since the current application got a non-null event. If it has been long enough, the VBL task blanks the screen and sets a flag indicating this fact. That's it.

### The GetNextEvent Filter

There is an undocumented "hook" in `GetNextEvent()` that allows special processing to be performed before control is returned to the calling application. In the global location `JGNEFilter` there is a pointer to a procedure that gets jumped-to just prior to returning to the application. In fact, the filter procedure completes with an RTS instruction which returns directly to the application.

When the filter procedure is entered, `A1` points to the event record in the application's address space. The event has been dequeued and copied into the application's event record. Finally, the top of the stack contains the address of the instruction following the application's `_GetNextEvent` trap, and just under that is the boolean result being returned by `GetNextEvent()`. Be aware that this information was obtained by digging with `MacsBug`, and may change without notice in future operating system revisions.

It may be of interest that the "real" filter procedure appears to perform the following services (there may be more):

- Checks for and beeps the alarm clock.

- Handles the special "command shift" keys, which eject disks, etc.

The CRT saver intercepts the `JGNEFilter` and keeps track of how long it has been since the application received a non-null event from `GetNextEvent()`. Then it jumps to the "real" filter procedure.

### INIT Resources

Each time the Mac is started up, the operating system installs ROM patches, loads keyboard maps and opens certain drivers. The mechanism used for this process is the INIT resource. You can make use of this feature of the Mac bootstrap code.

During startup, the boot code looks in the "System" file for up to 32 resources of type INIT, starting with `ID=1`. For each such resource found, the following steps are taken:

1. The INIT resource is loaded into the system heap.
2. `DetachResource()` is called, which "orphans" the resource, removing it from the map.
3. A JSR is made to the first location in the resource.
4. When the INIT code returns, via an RTS instruction, the first two locations in the INIT are bashed with NOP instructions.

This action may seem a little strange, so let's look at an INIT resource which installs a ROM patch.

First, the resource is loaded and detached, making it invisible to the Resource Manager. It is important to understand the need for detaching the resource. If you start an application on a new disk containing a System file, that disk becomes the new "system" disk. The current System file is closed, the System file on the new disk is opened, and all system resources start coming from the new System file.

When the old System file is closed, all resources that were loaded from there are released to make way for those in the new system. If the INIT resources were not detached after loading, they too would be released, with unfortunate results.

Once the INIT resource has been made permanently resident in the system heap, it is called at its first location, which is usually a jump to some one-time initialization code located at the *end* of the resource. This allows the initialization code to be chopped off with a `_SetHandleSize` after it is run, freeing up that system heap space.

The initialization code installs the ROM patch, computes the amount of space needed by the patch code only, then does a `_SetHandleSize` to reduce the resource's size in the system heap. Then it places a handle to itself in register `D7` and returns to the boot code.

At this point, the boot code assumes that there is no



need in the INIT resource for the jump to the initialization code, since it has run and may have been chopped off. So it uses that handle to bash the first 2 words of the INIT resource with No-Op instructions.

### The CRT Saver

The example C program is a CRT saver which gets installed via an INIT resource, uses a VBL task to time the screen blanking, and uses the GetNextEvent filter to keep track of the last time the application received a non-null event. Please remember that this is an example, written to present and illustrate ideas. In real life, this small program would be written entirely in assembler, and would use the "normal" installation method just explained.

The example shows an alternate installation method, where the initialization code copies the action code to a locked-down area in the system heap and then releases the entire INIT resource. There are two bugs in the CRT saver, which I have purposely left for you to solve.

The first one should be easy. If the application does not call GetNextEvent, the time of last non-null event does not get updated, and the screen may be prematurely blanked. Hint: use

the global variables MBTicks and KeyTime.

The other bug will be harder to fix (I have not solved it yet). The usual method for flashing a cursor on the screen (such as the TextEdit insertion bar) is to repeatedly XOR the pattern, which makes it flash white, then black. If the cursor happens to be white (invisible) at the moment the screen is blanked, the effects of succeeding XOR's are reversed. The cursor's manager thinks the cursor is white, yet it was secretly bashed black by the screen clearing process.

Because the XOR method is "open loop", the manager never finds out that the cursor's state was reversed. So when it's time to move the cursor, the manager tries to XOR it as needed to make it invisibly white, but instead leaves the space black. This has the effect of leaving behind a ghost of the cursor. Keep in mind that TextEdit isn't the only agent that uses flashing cursors.

The example uses specifics of the Apple 68000 Development System, and the Consulair Mac C compiler. You may need to take different approaches to producing the INIT resource and accessing static variables from both C and assembler. In addition, the interface into C from the Vertical Retrace Manager and the GetNextEvent filter may differ.

```
/*
 *
 * Screen Saver INIT Resource
 *
 * Written by: Robert B. Denny, Alisa Systems, Inc.
 * June, 1985
 * Written for: Apple Computer MDS Development System and Consulair Mac C™
 * This program uses specific features of the MDS system and Mac C
 * and will almost certainly require modifications for other systems.
 *
 * LINKER COMMAND FILE:
 * __
 * /OUTPUT Dev:CRTSave.INIT
 * /Globals -0
 * /Type 'RSRC'
 * /Resources
 * CRTSav
 *
 * $
 * __
 *
 * Copyright (C) 1985, MacTutor Magazine
 *
 * Permission granted to use only for non-commercial purposes. This notice must be
 * included in any copies made hereof. All rights otherwise reserved.
 *
 * Warning: This code was edited for publication, which could have introduced minor errors
 */
#Options R=4 /* Mac C: Use R4 to access globals */

#include "MacDefs.H" /* Basic Macintosh structures */
#include "Events.H" /* Event Manager & Event Record */
#include "OSMisc.H" /* Queue Elements and VBL defs */

#define TRUE 1
#define FALSE 0
```

```

#define NULL 0
#define SAVE_DELAY 18000 /* 5 Minutes in ticks. Saver delay */

/*
 * Definitions which allow easy to read access to system variables.
 */
#define Ticks ((unsigned long*)(0x16A)) /* Ticks since boot time */
#define GrayRgn ((unsigned long*)(0x9EE)) /* Desktop region w/rounded corners */
#define JGNEFilter ((ProcPtr*)(0x29A)) /* -> GetNextEvent filter procedure */

/*
 * Screen geometry parameters work on both 128K and 512K
 */
#define ScreenLow ((unsigned long*)(0x7A700)) /* -> base of screen map */
#define scrnLongwords (5472) /* No. of longwords in screen map */

/*
 * Static variables for the VBL Task and the GNEFilter
 */
VBLTask VBQElement; /* Vertical Retrace Queue Element */
unsigned long EvTicks; /* 'Ticks' value of last non-null event */
ProcPtr SavedJGNEFilter; /* Entry point of "normal" GNE filter proc */
unsigned short BlankScreenFlag; /* TRUE means screen already blanked */

/*
 * The following assembler code executes as called from the Mac boot process as an INIT
 * resource. Here is an example where C is just not appropriate.
 */
#asm

 RESOURCE 'INIT' 28 'CRT Saver' 80 ; System/Locked att's

 Include MacTraps.D
 Include SysEquX.D

 XDEF InitSaver

InitSaver:
 MOVE.L #(TheEnd-TheStart),D0 ; Allocate space for saver
 _NewPtr,SYS ; Locked, in system heap
 BNE @10 ; (not enough space)
 MOVE.L A0,A1 ; A1 -> destination area
 LEA TheStart,A0 ; A0 -> source area
 MOVE.L #(TheEnd-TheStart),D0 ; Size of code & static data
 _BlockMove ; Copy stuff to allocated block
 MOVE.L A1,A0 ; A0 -> Moved code & data
 LEA (OurStatics-TheStart)(A0),A0 ; A0 -> "Top" of real statics
 MOVE.L Ticks,EvTicks(A0) ; Initialize Last Event Ticks
 MOVE.L JGNEFilter,SavedJGNEFilter(A0) ; Save "real" GNE filter proc
 PEA (GNEIntfc-TheStart)(A1) ; Push -> to GNE filter interface
 MOVE.L (SP)+,JGNEFilter ; Now we catch GNE calls also
 LEA VBQElement(A0),A0 ; A0 -> VBL Queue Element
 PEA (VBLIntfc-TheStart)(A1) ; Push -> VBL Service Task
 MOVE.L (SP)+,vblAddr(A0) ; Fill in entry point in Q-Elem
 MOVE.W #Vtype,qType(A0) ; Indicate the queue element type
 MOVE.W #300,vblCount(A0) ; Schedule at 5 sec. intervals
 Move.W #5,vblPhase(A0) ; Off-the-wall phasing
 _VInstall ; Start the VBL task

;
; We finish up by disposing of ourselves.
;
@10:

```

```

 LEA InitSaver,A0 ; A0 -> Ourselves (INIT resource)
 MOVE.L A0,(A0) ; Make a handle to ourselves
 MOVE.L A0,D7 ; Save handle in D7 for MacBoot
 _RecoverHandle,SYS ; Get our real resource handle
 _DisposHandle ; Free ourselves
 RTS

#endasm

/*
 * The rest of this is copied into the allocated non-relocatable block and runs from there.
 * There are 2 interface routines in assembler which call C to do the real work, one for
 * the VBL task and the other for the GNE filter. Also, there is space allocated for our
 * local static variables.
 */
#asm

TheStart: ; Static data goes here via A4
 DCB.B 32,0 ; Enough room for statics

OurStatics: ; "Top" of static area

;
; Interface to C for Vertical Blanking Task
;
VBLIntfc: ; Interface for VBL task service
 MOVEM.L A4-A5/D4-D7,-(SP) ; Save registers
 LEA OurStatics,A4 ; A4 -> Our static variable area
 JSR VBLRoutine ; Call C for dirty work
 MOVEM.L (SP)+,A4-A5/D4-D7 ; Restore registers
 RTS

;
; Interface to C for GetNextEvent filter. This must finish up
; by jumping to the "real" filter whose address was originally
; in the global "JGNEFilter".
;
GNEIntfc:
 MOVEM.L A1-A5/D0-D7,-(SP) ; Be safe.
 LEA OurStatics,A4 ; A4 -> Our static variable area
 MOVE.L A1,D0 ; Pass -> Event Record as parameter to C
 JSR GNEFilter ; Call C to filter the events, etc.
 MOVE.L SavedJGNEFilter(A4),A0 ; Where to next?
 MOVEM.L (SP)+,A1-A5/D0-D7 ; Restore saved reg's
 JMP (A0) ; Jump to "real" GNE filter

#endasm

/*
 * Vertical Blanking task. This is periodically rescheduled and handles watching for
 * no-activity intervals and blanking the screen if more than a specified time elapses
 * between non-null events.
 */
VBLRoutine()
{
 unsigned long *lp;
 unsigned long n;
 extern GNEIntfc(); /* Declare this as a proc */

 VBQElement.vblCount = 300; /* Reschedule ourselves */

/*
 * If the screen is already blank, do nothing.
 * You might enhance this by doing something interesting with the cursor inside
 * this "if" statement, knowing you'll get here every 5 seconds.
 */

```

```

 */
 if(BlankScreenFlag)
 {
 return;
 }
 /*
 * Blank the screen if there hasn't been a non-null event in the last SAVE_DELAY ticks.
 */
 if((Ticks - EvTicks) > SAVE_DELAY)
 {
 BlankScreenFlag = TRUE; /* Set the blanked flag */
 HideCursor (); /* Hide the cursor from blanking */
 lp = ScreenLow;
 for(n=0; n<scrnLongwords; n++) /* Blank the screen */
 *lp++ = 0xFFFFFFFF;
 ShowCursor (); /* Restore the cursor */
 }
 return;
 }

 /*
 * GNE Filter Procedure
 *
 * If there has been either a key press or mouse click since the screen was blank, restore the
 * screen and switch the GNEFilter back to normal.
 */
 ProcPtr GNEFilter(ep)
 EventRecord *ep; /* -> Event Record just dequeued */
 {
 if(ep->what != nullEvent) /* If appl is getting non-null event */
 {
 EvTicks = Ticks; /* Remember ticks at non-null event */
 if(BlankScreenFlag) /* If the screen is black */
 {
 BlankScreenFlag = FALSE; /* It's going to get refreshed */
 DrawMenuBar (); /* Draw the menu bar, then */
 PaintBehind (FrontWindow (), GrayRgn); /* Draw all windows and desktop */
 }
 }
 return;
 }

 #asm
 TheEnd: ; Mark the end of the resident code/data
 ;
 ; ----- END OF CODE & DATA WHICH IS COPIED TO ALLOCATED BLOCK -----
 ;
 #endasm

```



# Toolbox Tips

## Arrows from C



Rick Flott  
Chandler, AZ  
MacTutor Vol. 1 No. 9

### How to Draw Arrowheaded Lines

While writing a new application, I discovered I needed to have lines with arrowheads (like the ones we all love in MacDraw). Not only that, I needed to have the line stop on the surface of an object (usually an oval or rectangle).

The first place to look, of course, is to see how MacDraw did it. After using MacDraw for some time in various applications, I was surprised to see that the arrowheads were not 3-pointed polygons but actually small wedges! This was discovered after placing an arrowhead on a thick line and noticing the curvature of the arrowhead.

Keeping with the Macintosh spirit, I decided to make the call similar to the ones used in QuickDraw (i.e. Line and LineTo). Instead of using a LineTo call, this routine may be substituted and all of the LineTo's changed to ArrowLineTo's.

The parameter passed to it is the absolute horizontal and vertical coordinates of the end point of the line (as in LineTo). The end point is also where the arrowhead will be drawn.

The basic algorithm is simple -

- 1). Compute the slope of the line
- 2). Draw the line
- 3). Point the wedge in the opposite direction of the line (to create the effect of an arrowhead).
- 4). Draw the wedge

There is a fantastic Mac routine called PtToAngle that will compute the angle for you! All you need to do is to place a rectangle around the starting point, make the call, and the angle is returned to you (in degrees)!

The following code (written in Mac C) is pretty much self explanatory. You could experiment with the constants arrowWidth and arrowLength (as shown in Fig. 1) to vary the width and length of the arrowhead. Note: this routine could be changed so that these constants are passed as parameters for applications that need varying sizes of arrowheads.

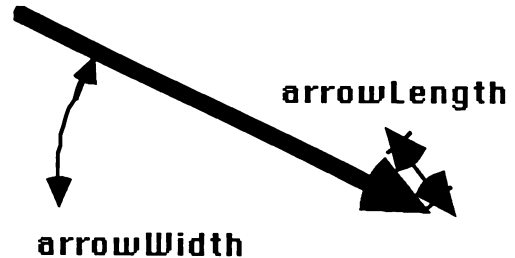


Fig. 1 Arrow Constants

```
/*.....
 ArrowLineTo.c
 Copyright 1985, FlottWare for MacTutor
 */

#include "MacCDefs.h" // Mac ROM data structure def
#include "Events.h"
#include "Window.h"

extern struct P_Str *CtoPstr();
int strlen(str) char *str;
{int i=0; while (str[i++]); return i-1;}

/*-----
Global Data
-----*/

#define False 0
#define True 0xFF

Rect windowRect = {40,4,336,508}; // Window Rect
WindowPtr windowPtr; // Window Pointer
EventRecord event; // Current event record

/*-----
main()
-----*/

main()
{

if (CatchSignal()) // Is the user quitting this application?
ExitToShell(); // Go to Finder when done

init(); // Init Mac
```

```

while (True) // Poll events forever
{
 SystemTask(); // Perform System duties

 if (GetNextEvent(everyEvent, &event)) // Get next event
 {
 switch (event.what) // Which event is it?
 {
 case autoKey:
 case keyDown: // Key was pressed
 case mouseDown: // Mouse Button pressed
 {
 Signal("All Done"); // Y - quit application
 break;
 } // case autoKey,keyDown,mouseDown:

 } // switch (event.what)
 } // if (Getevent)
 } // while (true)
} // main

/*****
Windows
*****/

/*-----
init() -
-----*/

init()
{
 // Open a window

 windowPtr = NewWindow(0,&windowRect,CtoPstr("Hit Key or
Mouse to Quit..."),True, documentProc,-1,True,0);

 SetPort(windowPtr);

#define centerH 250
#define centerV 150
#define offset 50

 // Draw some lines with arrowheads

 MoveTo(centerH,centerV);
 ArrowLineTo(centerH-offset,centerV);

 MoveTo(centerH,centerV);
 ArrowLineTo(centerH+offset,centerV+offset);

 MoveTo(centerH,centerV);
 ArrowLineTo(centerH,centerV+offset);

 MoveTo(centerH,centerV);
 ArrowLineTo(centerH,centerV+offset);

 MoveTo(centerH,centerV);
 ArrowLineTo(centerH,centerV+offset);

```

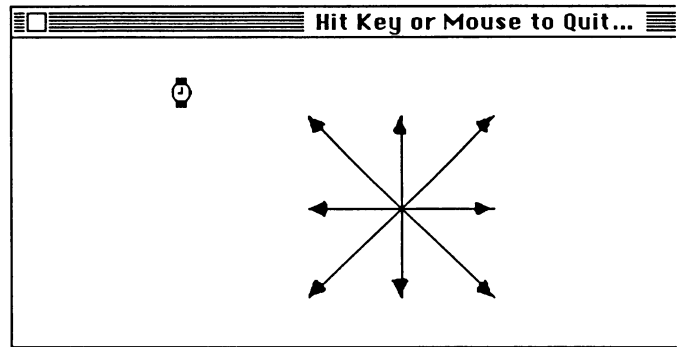


**Fig. 2 C Program Output**

```

ArrowLineTo(centerH+offset,centerV+offset);

```

```

MoveTo(centerH,centerV);
ArrowLineTo(centerH+offset,centerV);

```

```

MoveTo(centerH,centerV);
ArrowLineTo(centerH+offset,centerV-offset);

```

```

MoveTo(centerH,centerV);
ArrowLineTo(centerH,centerV-offset);

```

```

MoveTo(centerH,centerV);
ArrowLineTo(centerH-offset,centerV-offset);

```

```

} // end init()

```

```

/*-----
ArrowLineTo(horiz,vert)

```

This routine draws a line from the current pen location to point (h,v) with an arrowhead (wedge) on the end of it.

Inputs : horiz - horizontal coord of end point  
vert - vertical coord of end point

Written by: Rick Flott Mac C (Consulair) V 1.0  
-----\*/

```

ArrowLineTo(horiz,vert)
short horiz;
short vert;
{
 // Size of starting Rect used in PtToAngle
#define rectOffset 30
 // Arrowhead (wedge) constants
 // arrowWidth - width of 1/2 arrowhead (in degrees)
 // arrowLength - length of arrowhead in pixels
#define arrowWidth 25
#define arrowLength 10

 // Rects used in angle calculations
 Rect startRect; // starting Rect used in PtToAngle

```

```

Rect arrowheadRect;
// the Rect that the wedge is drawn in

short arrowAngle; // Angle of arrowhead (in degrees)

Point startPt,endPt; // Start,End points of line

GetPen(&startPt);
 // Get the current pen location
 // Set up a rectangle around the starting point
 // (this is needed for the PtToAngle routine)

SetRect (&startRect,startPt.h - rectOffset,
 startPt.v - rectOffset,
 startPt.h + rectOffset,
 startPt.v + rectOffset);

SetPt(&endPt,horiz,vert); // Set up the end point
 // Calculate the angle (in degrees) of the line segment

PtToAngle(&startRect,&endPt,&arrowAngle);

LineTo(horiz,vert); // Draw the line

arrowAngle -= (180 + arrowWidth);

```

```

// Create a arrowhead with a wedge facing opposite direction
// Set up a Rect for the wedge around the endpoint
SetRect (&arrowheadRect,endPt.h - arrowLength,
 endPt.v - arrowLength,
 endPt.h + arrowLength,
 endPt.v + arrowLength);

 // Draw arrow head (a reversed wedge)
PaintArc(&arrowheadRect,arrowAngle,2*arrowWidth);

} // end ArrowLineTo()

```

```

!start
]
(
macclib
ArrowLineTo

```

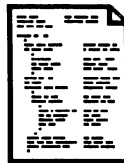
**Fig. 3 Linker File**



# C Workshop

## Dial-a-Fortune Apple Talk Server

C



Robert B. Denny  
Timothy T. Coad  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 10

It's been about six months since the AppleTalk network was released. Yet there seems to be little software available which uses this network. One reason is that the market is too small as yet and developers can't see enough potential to risk the investment. Another reason is that AppleTalk applications appear difficult to implement, requiring a great deal of knowledge and understanding that is not easy to obtain. Last, but certainly not least, Apple has failed to complete the basic services that history has shown are required for successful introduction of a computer network.

About six or seven months ago, Apple released a publication called Inside AppleTalk, available from the same source as Inside Macintosh. In addition, most developers who subscribed to the Apple "Software Supplement" have received The AppleTalk Manager: A Programmer's Guide, a part of Inside AppleTalk. These documents do a fairly good job of describing the network architecture and the details of using the AppleTalk services. If you have no experience with local area networks, however, you may get lost in the jargon and fail to "see the forest through the trees".

This month's C Workshop is devoted to AppleTalk. It includes a complete server application called *Confucious* which, when run on a Mac connected to the network, provides a "guru" service. A Mac Pascal client application is described by Alan Wootton in a separate article in this issue of Mac Tutor. The Pascal client, running on another Mac, locates the C server and makes requests for pearls of wisdom, which the server gladly sends back.

In order to keep this article's size reasonable, we assume that (1) you understand how to use the Mac's I/O services and (2) you have access to (at least) the AppleTalk Manager Programmer's Guide. The article describes the overall architecture of the AppleTalk network, and concludes with a listing of the server's code, written mostly in C. We'll look at programming details next month.

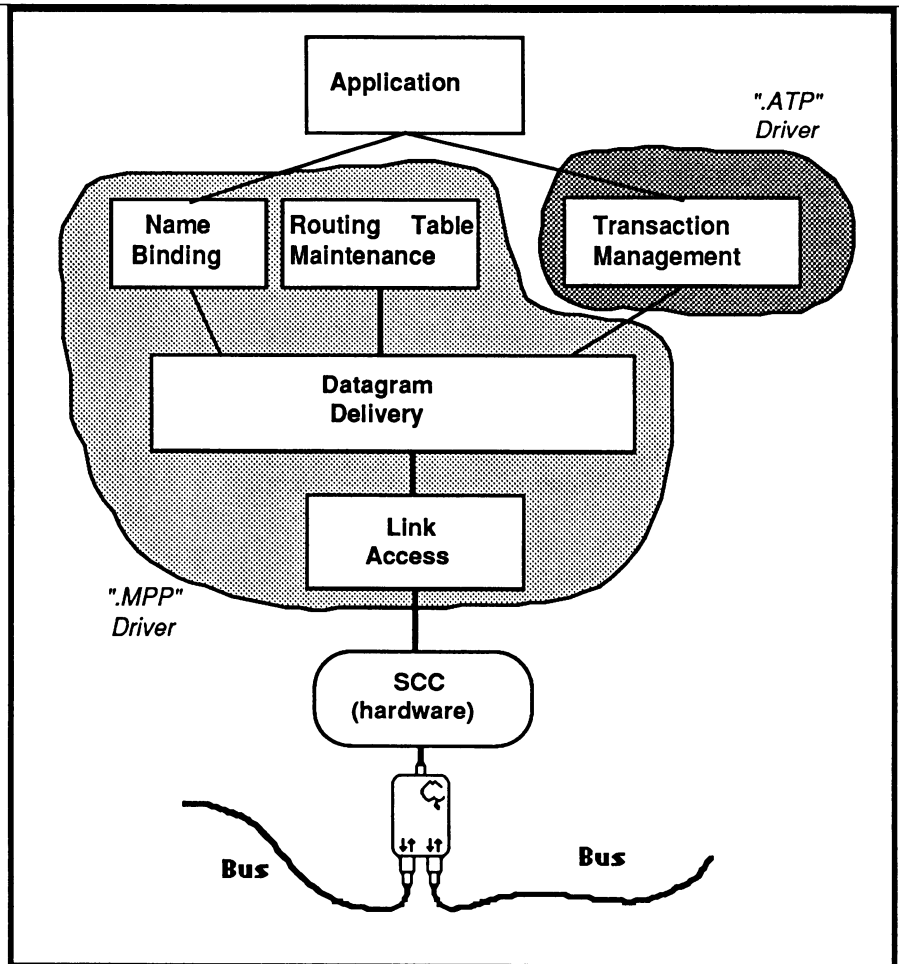


Fig. 1 - AppleTalk Components

### NOTE

The descriptions given in this article are simplified so as to make it easier to understand the overall architecture of AppleTalk. Please consult the Apple documentation for full details.

### Protocols and Layers

The lifeblood of any network is its suite of *protocols* that are used to negotiate and communicate. Usually, there are two *agents* in a network operation, the *client* and the *server*. The operation consists of a set of information exchanges between the client and the server, according to a protocol.

In order to bring some order to the chaos that would be created by a multitude of protocols, some of which are clients of other protocols and servers to yet others, network designers use the concepts of *layers* and *interfaces*. A layer implements a set of services via an interface. For example, the file system on



the Mac has a layer that handles "raw" disk I/O (the device manager), and a layer on top of that which handles the organization of data on the disk into files and directories (the file manager). This separation of functions makes it possible to handle both hard disks and floppies with the same file manager routines. The file manager uses a defined interface to the device manager for its requests. Finally, the organization of data on the disk as files and directories is a sort of "spatial protocol".

Network architectures are normally divided into seven layers, according to the ISO "Open Systems Interconnection" convention adopted about five years ago. These layers, from lowest to highest are:

|                      |                                                                                                                                 |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <b>Physical:</b>     | Electrical and mechanical aspects of communication circuits.                                                                    |
| <b>Data Link:</b>    | Data and frame formats, bus access & contention protocols, error detection, other low level link services.                      |
| <b>Network:</b>      | Addressing and routing of messages between "sockets" transparently across multiple data links.                                  |
| <b>Transport:</b>    | Error and flow control between sockets. Exchanges of information between routing agents on special sockets.                     |
| <b>Session:</b>      | Opening and closing logical links between sockets. Starting up remote servers on request. Naming of entities and name services. |
| <b>Presentation:</b> | Cryptosystems, code conversion (e.g., ASCII to EBCDIC), other transformations needed for the native application environment.    |
| <b>Application:</b>  | Protocols used between specific applications.                                                                                   |

### AppleTalk Architecture

Apple Computer has so far defined and published protocols for the physical through the transport layers of AppleTalk. Generally speaking, AppleTalk has a bus oriented, packet-based architecture, supporting internetworking, named objects, adaptive routing and a transaction-oriented transport service.

Services are implemented in a pair of I/O drivers and associated resources which are opened permanently on 512K Macs, and live in the system heap. On 128K Macs, applications which want to use the network must manually open the drivers, loading them into the application heap. This means that no "detached" service processes in the system heap can be supported on the smaller systems. Figure 1 shows the breakdown of AppleTalk functions and the drivers that implement them.

#### Physical & Data Link Layers: ALAP

The physical layer consists of a bus topology twisted-pair cable carrying electrical signals conforming to the EIA RS-

422 standard. The Mac hardware has a "serial communications controller" (SCC) which is capable of being software configured as a standard async or sync port, or as an AppleTalk port. AppleTalk uses the printer port on the Mac.

The data link layer is implemented in software and uses the "AppleTalk Link Access Protocol" (ALAP) to manage usage of the common bus and to detect transmission errors between physically connected Mac systems. ALAP uses a frame format which is similar to that used by several well known block-oriented protocols such as HDLC. The algorithms used to manage multiple access to the bus are specific to AppleTalk, and fall into the category of "Carrier Sense Multiple Access" (CSMA).

Simply stated, the rules of CSMA are: (1) talk if no one else is talking; (2) if you want to talk and some one else is already talking, go away and try again later; (3) if two of you start talking at the same time, yell "Abort!" so everyone else knows, then both of you go away and try again later.

One other thing to note: The assignment of *node numbers* to each station on a network is not done by the user. Rather, the ALAP software tries to assign a random node number when the Mac is booted, then shouts it out over the net. If no one else objects, that is the node number to be used. Otherwise, another try is made. As an application developer, you'll rarely deal directly with ALAP.

#### Network Layer: DDP

The network layer uses the "Datagram Delivery Protocol" (DDP), which defines the format of the address header which is included in each packet and which stays with the packet between cooperating processes through communication endpoints called *sockets*. The path between sockets may span several physical links, and may go through intermediate systems. One Mac may have several AppleTalk connections open at a time, so the *node* number is not enough to identify a network address.

An address must consist of at least a node number and a socket number. In fact, AppleTalk considers a socket to be the one and only *network-visible entity*. Socket numbers fall into two groups. The "well-known" sockets (1-127) are assigned by Apple for specific generic network services and experimentation. The others (128-254) are called "dynamic" sockets. They are assigned temporarily and are often associated with a name managed by the Name Binding service of AppleTalk (see below).

Individual AppleTalk networks may be tied together to form an *internetwork*. Node numbers are unique only within a single physical network, so DDP requires that each network be assigned a *network number*. Figure 2 (see top of next page) shows a three-net internetwork. The Networks are connected by either a full *bridge* or a pair of *half-bridges*. The full bridge attaches directly to two physical networks and provides datagram routing from one to the other. The pair of half bridges does the same thing, but they use an intermediate transmission medium such as a long-distance telephone line.

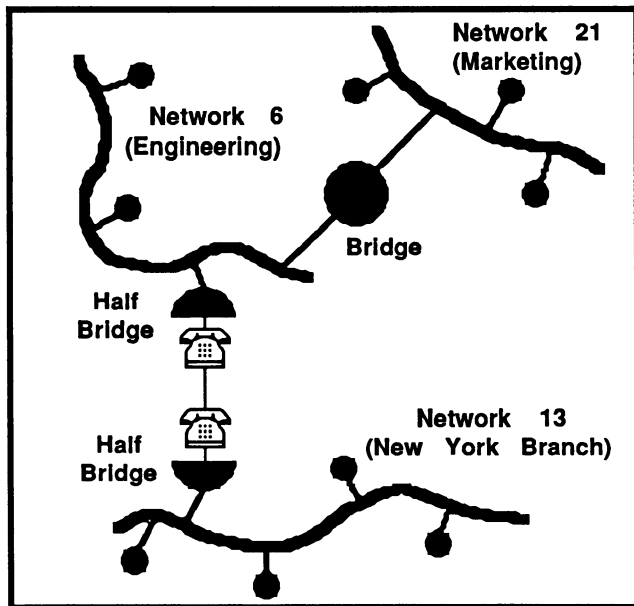


Fig. 2 - An AppleTalk Internetwork

DDP defines an internet address consisting of a net number, a node number and a socket number. One might argue that the net number is superfluous, since the node number could be made unique within the entire internet. True, but the early local net designers at Xerox PARC decided that routing overhead should be minimized.

The key to their idea is that many nodes are attached to a single network, and that getting the packet to the proper local network is all that's needed to get it to the destination node. Normally, there are far fewer networks than nodes, so the routers have a much easier task. AppleTalk incorporates this *network-oriented* routing system.

The basic service provided by DDP is to "try its best" to get a packet from the source to the destination. If there is a problem somewhere, the packet is vaporized. DDP doesn't care about the details of link management or signalling, it only needs some sort of link service to its neighbors in the net. Application developers rarely use the datagram services directly.

### Routing Services: RTMP

The DDP internet routing process requires information as to the topology (interconnect configuration) of the internet. Each router involved in getting a DDP datagram from the source to the destination net must know where to send the datagram next. The routers use a set of *routing tables* to maintain this information.

The tables consist of a list of net numbers paired with the local node number of the bridge through which the shortest path to that net exists. These tables are dynamically constructed and maintained, with no need for a network manager to describe the topology. Automatic routing table maintenance is done by the routers carrying on a continuous

chit-chat, proclaiming themselves as routers and broadcasting routing information for other routers, using the *Routing Table Maintenance Protocol* (RTMP).

Routing information exchange uses DDP at its communication medium, via socket number 1. Each Mac has a small RTMP-speaking process that keeps track of the nearest router and the network number of the local net. Bridges have a full-function RTMP-speaking process which engages in the prescribed chit-chat to maintain their routing tables. Bridges use this routing table to relay DDP datagrams according to their destination network number.

### Transport Layer: ATP

The transport layer protocols are used to control the flow of data between sockets, and for end-to-end error recovery. Clients of the transport layer are offered some sort of "logical link" or "virtual circuit" service between themselves and other network visible entities.

Most networks are fairly similar in the first three layers, up through the network layer. Above that, however, their architectures begin to diverge. The transport layer is usually the first to reflect the main purpose for which the network was designed. AppleTalk is quite unusual in this respect.

Typically, the transport layer supports a two-way link between clients. Some sort of higher level protocol handles the opening of the link, and the transport layer provides error and flow control. The clients at each end can act as though they have an error-free hardwired line between them. AppleTalk does not support this sort of protocol.

Instead, AppleTalk has a *transaction-oriented* transport protocol, called AppleTalk Transaction Protocol (ATP). Here, one end acts as a *requestor* and the other acts as a *responder*. The responder usually sets itself up on a "permanent" socket. Requestors can "pop up" anywhere, at any time, and send requests. The example application at the end of this article is a responder.

The requestor sends a request which may carry with it up to 578 bytes of data in a single *segment*. The responder may send back up to eight such segments, for 4624 bytes total. If the request arrives in error, it is simply discarded. The requestor keeps trying until it receives a response. Response segments that are damaged are re-transmitted. This selective retransmission is handled by the ATP protocol, invisible to ATP clients.

Sometimes it is important that a request be serviced once and only once. An example of such a request would be to backspace a record in a remote file. Remember that the requestor repeatedly requests until it receives a response.

ATP provides a special class of transaction service called "exactly once". Each request is tagged with a *transaction ID*. The responder ATP has the logic needed to assure that the request with a given transaction ID is executed only once. The penalty for such service is extra overhead.

*Don't mess with part time Mac coverage.  
Get professional with the developer's  
choice: Subscribe to MacTutor today!*

## Named Entities: NBP

Recall that the only network-visible entities are sockets (socket clients, actually). Communication between processes on AppleTalk takes place between socket clients. A key feature of the network is that most objects are accessible by name rather than by address. Node numbers and dynamic socket numbers are assigned randomly. People prefer names.

The Name Binding Protocol (NBP) defines a set of services that applications can use to register names for their sockets, and to locate named entities. AppleTalk names consist of three fields: the *object* name, the *type* name and the *zone* name. The object name is the "name" as we usually think of it. The type indicates what sort of object it is. The zone field is not currently supported by AppleTalk and is always set to the value "\*", meaning "this zone". The format of a name is:

**object:type@zone**

The object and type fields in a name may contain the wildcard character "=" which stands for "all possible values". Thus the name "=:printer@" means "all printers in this zone", or simply "all printers" since the zone concept is not yet supported.

Usually, a server will make itself available to the network by opening an ATP-responding socket, then registering itself by name, associating the name with its network address (net/node/socket). Then a client may lookup the server by name, or it may look up all servers of that type (using the "=" wildcard) then choose a particular one for use.

## Limitations of AppleTalk

Now that we have an overview of the network architecture, let's look at the limitations of AppleTalk, especially its unusual transport layer protocol, ATP. Understanding these limitations is very important to you, as a developer. Unless Apple faces up to these limitations, you will be saddled with the responsibility of creating your own "session" oriented service, either on top of ATP or in addition to it. Let's see why.

Recall that ATP provides a transaction-oriented service. There is no provision for sequencing or otherwise grouping transactions. There is no "private wire" service between cooperating applications.

Consider a file server, whose ATP responding socket is visible to the whole network. Anyone, at any time, is free to issue an ATP request to the server's socket, and to expect a response. But file service implies a "context" spanning several transactions: an open file with a current file position.

This means that the file server must remember what's going on for each client who has a file open on that server. And a single client may have more than one file open, further complicating matters. The client and server need to set up a conversation which will continue across several transactions.

An open file is typical of a general class of network

operations which require a private "session" to be maintained between the client and the server. This is the domain of the ISO session layer. AppleTalk has no session layer support.

You may say, "Well, at least ATP handles error and flow control; I can build sessions myself". But layering a session protocol on top of a "reliable transaction" protocol creates other problems. If we believe in internetworking, systems of interconnected AppleTalk bus networks, then we must face the problem of *transport delay*.

Suppose we want to set up an internet session, between sockets on nodes which are on different physical networks, separated by several bridges and perhaps a backbone network. The connection between client and server has plenty of speed capacity, but packets may be delayed up to 100 milliseconds in bridges and routers. ATP requires completion of a transaction before another can start. With a 100 mS delay from one end to the other, our transport protocol is capable of one transaction every 200 mS, or 5 transactions per second! Without the transport delay, the Macintosh ATP implementation is capable of hundreds of transactions per second.



Figure 3 - Server Dialog Box

It is important to understand that the degradation is *not* due to speed limits of the transmission medium, but instead due to the transport delay from one end to the other and ATP's need to complete a transaction before the next can be started. In an internetworking environment such as AppleTalk, transport delays are a fact of life. ATP is poorly suited as a transport protocol in such situations.

Another aspect of ATP is its *asymmetry*. An ATP request may carry up to 578 bytes of data. The response, on the other hand, can carry 8 times that amount of data. The responding socket is more or less permanent, the requesting socket is opened and closed for each request. The responder has no way of delivering an asynchronous message to the requestor. Data flow from the requestor to the responder has a higher overhead than that in the opposite direction.

These facts have caused some developers to set up responding sockets at each end of a client-server connection. This approach uses two sockets at each end (temporary

requestor, permanent responder). But there is a limit to the total number of ATP sockets that may be open in the current version of AppleTalk, 6 requestors and 6 responders. So sockets are a precious quantity and this approach wastes sockets unnecessarily.

AppleTalk has no session layer, a severe architectural deficiency which will promote multiple "private" session protocols, some of which may be poorly designed for the internet environment. Session protocols are among the most difficult to design and test. Often, the theoretical behavior of the network used as the basis for protocol design turns out to be inaccurate. Error recovery in the session layer is probably the most complex design issue in networking.

There is another limitation that took me by surprise. AppleTalk does not support transactions between sockets on the same node. At first this might not seem to be too serious but, as any networking veteran will tell you, it is serious. If your node happens to be the one with a server you need, you can't get to it. Keep this in mind when designing AppleTalk applications: your application must handle the case where the server and client are on the same node.

### A Sample AppleTalk Server

The rest of this article contains the sources for an example AppleTalk "Dial-A-Fortune" Server. Note that the main program is fairly simple, and that most of the AppleTalk access functions are broken out separately and written for general use. (See fig. 3 on the previous page.)

There is not room for explanation of the details, but much of the information you'll need to understand this program is covered in Alan Wootton's article in this issue. I will explain its operation next month, along with more about AppleTalk.

One last item: You do NOT have a copy of the file ATALK.H, which contains the structure definitions and constant definitions needed for AppleTalk I/O. The items required for this program will be published next month. The full ATALK.H will be available on a MacTutor Source Disk. Also, look for some interesting AppleTalk developer's tools from Consulair soon.

### Resource File In RMaker Format

```
*
* DFServer.R - RMAKER source.
*
Dev:DFServer
APPL????
* Load linker output
Include Dev:DFServer.Rsrc
*
* Define the DIAL-A-FORTUNE SERVER Modal dialog.
*
Type DLOG
,128
No Title
40 96 240 416
Visible
1
```

```
0
128
```

```
Type DITL
,128
6
```

```
Button
175 20 195 90
Fortune
```

```
Button
175 230 195 300
Quit
```

```
StaticText Disabled
5 80 20 310
AppleTalk Dial-A-Fortune
```

```
StaticText Disabled
20 72 35 310
^0
```

```
StaticText Disabled
45 110 60 310
Confucious Say:
```

```
StaticText Disabled
70 20 170 310
Press FORTUNE to get your Fortune or QUIT to ++ shutdown
the AppleTalk Dial-A-Fortune Server.
```

```
*
* Entity string components
*
```

```
Type STR
,128
Confucious
```

```
Type STR
,129
Dial-A-Fortune
```

```
Type STR
,130
\2A
```

```
*
* Number of fortunes and the fortunes themselves.
*
```

```
TYPE DATA = GNRL
,128
.H
nnnn
```

```
Type STR
,131
One of the greatest sources of energy is pride in ++
what you are doing.
```

Type STR  
,132  
Take a cannibal to lunch.

( more fortune strings ...)

Type STR  
,150  
Every successful person has had failures but repeated failure  
is no ++  
guarantee of eventual success.

END

```
/*
 * Dial-A-Fortune AppleTalk™ Server Application
 *
 * Written by: Timothy T. Coad, Alisa Systems, Inc.
 * June, 1985
 * Written for: Apple Computer MDS Development System and Consulair Mac C™
 * This program uses specific features of the MDS system and Mac C
 * and will almost certainly require modifications for other systems.
 *
 * LINKER COMMAND FILE (uses Mac C minimum library, output is read by RMaker)
 * __
 * IStart
 *
 * /Output Dev:DFServer.Rsrc
 * /Type 'RSRC'
 *
 * MacCLib
 * DFServer
 *
 * $
 * __
 *
 * Copyright (C) 1985, MacTutor Magazine
 *
 * Permission granted to use only for non-commercial purposes. This notice must be
 * included in any copies made hereof. All rights otherwise reserved.
 *
 * Warning: This code was edited for publication and may possibly contain minor errors
 */

#define VERSION "032Experimental version 1.0.A"

#include "Lib:MacDefs.H" /* The usual collection of Mac definitions */
#include "Lib:QuickDraw.H"
#include "Lib:Window.h"
#include "Lib:TextEdit.H"
#include "Lib:Dialog.h"
#include "Lib:Events.H"
#include "Lib:PBDefs.H"

#include "Lib:ATalk.H" /* AppleTalk definitions to be published next month */

#define FALSE 0
#define TRUE 1
```

```

#define DIALOG_ID 128
#define FORTUNE_DI 6 /* Dialog Item number of Fortune item */

#define NUMOFSTRINGS_ID 128 /* Resource ID of fortune string count ('DATA') */
#define FIRSTSTR_ID 131 /* Resource ID of first fortune string ('STR') */

/*
 * Resource IDs of the three parts of server's Network Entity Name.
 */
#define OBJECT_ID 128 /* Object name (Confucious) */
#define TYPE_ID 129 /* Object type (Dial-A-Fortune) */
#define ZONE_ID 130 /* Our zone ('' = 'This zone') */

/*
 * Global Variables
 */

NBPElem *ourName; /* -> Our registered name/skt (locked block) */
ATPBlock ATPGetReq, ATPResp; /* Statically allocated I/O parameter blocks */
BDS RespBDS; /* Buffer Descriptor Structure for response */
char ReqPkt[256]; /* Buffer to receive incoming requests */
char RespPkt[256]; /* Buffer for responses (fortune text) */
unsigned short NumOfStrings; /* Number of fortune string resources */
Handle Fortuneitem; /* Handle to currently selected fortune string */

extern short atpOpenSkt();
extern short atpCloseSkt();
extern int GetReqComplete();
extern int FilterGlue();

/*
 * Dial-A-Fortune Main Program
 */
main()
{
 unsigned short button, i; /* Locals, usage is obvious except as noted */
 Handle handle;
 DialogPtr dlog;
 unsigned short type;
 Rect rect;
 char fortune[256];

 InitDialogs (0); /* No restart function */
 InitCursor (); /* Make arrow cursor */
 SaveA5(); /* Save our A5 for async completion routines */

 /*
 * Allocate a non-relocatable NBPElem structure, request for an ephemeral responding socket
 * and register ourselves on the net.
 */
 ourName = (NBPElem *)NewPtr (sizeof(NBPElem));
 ourName->tuple.addr.skt = 0;
 ourName->tuple.addr.net = 0;
 if(atpOpenSkt(&(ourName->tuple.addr.skt)))
 {
 DisposPtr (ourName); /* Can't open a socket */
 return; /* Release tuple storage */
 /* and exit. */
 }

 /*
 * Get the three parts of the Network Entity Name and register ourselves using NBP.
 */
 handle = (Handle)GetResource ('STR ', OBJECT_ID);

```

```

i = pstrcpy(ourName->tuple.entity, *handle);
ReleaseResource (handle);
handle = (Handle)GetResource ('STR ', TYPE_ID);
i += pstrcpy(ourName->tuple.entity + i, *handle);
ReleaseResource (handle);
handle = (Handle)GetResource ('STR ', ZONE_ID);
pstrcpy(ourName->tuple.entity + i, *handle);
ReleaseResource (handle);
if(NBPPRegister(2, 8, ourName))
/* Registration error? */
{
 atpCloseSkt(ourName->tuple.addr.skt);
 DisposPtr (ourName);
 return;
/* Yes, free the socket and ... */
/* ... release tuple storage. */
/* Exit */
}

/*
 * The number of fortunes is stored in a resource to make it possible to add new ones.
 */
handle = (Handle)GetResource ('DATA', NUMOFSTRINGS_ID);
NumOfStrings = (unsigned short)((unsigned short)(*handle));
ReleaseResource (handle);

/*
 * Prime the pump with the initial asynchronous ATP Get Request.
 */
GetATPRequest(&ATPGetReq, ourName->tuple.addr.skt, ReqPkt, 256, GetReqComplete);

/*
 * Put up our dialog box. Use a modal dialog filter procedure to handle the completion of the
 * GetATPRequest. The GetATPRequest completion routine posts an event which is filtered
 * and processed by the modaldialog filter procedure.
 */
dlog = (DialogPtr)GetNewDialog (DIALOG_ID, 0, -1);
GetDItem (dlog, FORTUNE_DI, &type, &Fortuneitem, &rect);
ParamText (VERSION, "", "", "");
button = 0;
while(button != cancel)
{
 ModalDialog (FilterGlue, &button);
 if(button == oK)
/* Fortune button pressed? */
/* Yes, put up a fortune */
 {
 GetFortune(fortune);
 Set/Text (Fortuneitem, fortune);
 }
}
DisposeDialog (dlog);
/* Dispose of dialog */
NBPPRemove(ourName->tuple.entity);
/* Remove us from net */
atpCloseSkt(ourName->tuple.addr.skt);
/* Free the socket */
}

/*
 * SendFortune
 *
 * Get a random STR resource, display in the server's dialog box, ship it back to the
 * requestor as an ATP response packet and finally re-arm ourselves for another request.
 */
SendFortune()
{
 GetFortune(RespPkt);
 Set/Text (Fortuneitem, RespPkt);
 SendATPResp(&ATPResp, &ATPGetReq.csParam.atpGetRequest.addr,
 ourName->tuple.addr.skt, RespPkt, 256, &RespBDS,
 ATPGetReq.csParam.atpGetRequest.transID, 0);
}

```

```

 GetATPRequest(&ATPGetReq, ourName->tuple.addr.skt, ReqPkt, 256, GetReqComplete);
}

/*
 * GetFortune
 *
 * Get a random STR resource, modulo number of strings. Return pointer to string in memory.
 */
GetFortune(fortune)
char *fortune;
{
 Handle string;
 unsigned short index;

 index = FIRSTSTR_ID + ((unsigned short)Random ()) % NumOfStrings;
 string = (Handle)GetResource ('STR ', index);
 pstrcpy(fortune, *string);
 ReleaseResource (string);
}

/*
 * AppleTalk Name Binding Protocol (NBP) Low Level I/O Routines
 */

/*
 * nbpRegister - Register an element with AppleTalk.
 *
 * Returns the I/O result.
 */
short nbpRegister(interval, count, elem)
unsigned char interval;
unsigned char count;
NBPElem *elem;
{
 MPPBlock block;

 block.ioRefNum = mppRefNum;
 block.csCode = _nbpRegister;
 block.csParam.nbpRegister.retry.interval = interval;
 block.csParam.nbpRegister.retry.count = count;
 block.csParam.nbpRegister.verify = 1;
 block.csParam.nbpRegister.elem = elem;
 return(PBControl(&block, 0));
}

/* I/O is to the MPP driver */
/* Function is "Register name" */
/* Fill in retry interval */
/* Fill in retry count */
/* Enable name clash checking */
/* Fill in -> name to register */
/* DO IT, return the result */

/*
 * nbpRemove - Remove an entity from AppleTalk Registry.
 *
 * Return the I/O result code.
 */
short nbpRemove(entity)
Entity *entity;
{
 MPPBlock block;

 block.ioRefNum = mppRefNum;
 block.csCode = _nbpRemove;
 block.csParam.nbpRemove.entity = entity;
 return(PBControl(&block, 0));
}

/* I/O to .MPP driver */
/* Function is "Remove name" */
/* Fill in -> to entity string */
/* DO IT and return result */

```



```

/*
 * AppleTalk Transaction Protocol (ATP) Low Level I/O Routines.
 */

/*
 * atpOpenSkt - Open an dynamic ATP responding socket.
 *
 * Socket number set to 0 if it fails. Returns the I/O result code.
 */
short atpOpenSkt(skt)
unsigned char *skt; /* -> byte to receive assigned socket number */
{
 ATPBlock block; /* I/O Parameter block */

 block.ioRefNum = atpRefNum; /* I/O is for .ATP driver */
 block.csCode = _atpOpenSkt; /* Func is "Open Responding Socket" */
 block.csParam.atpOpenSkt.skt = 0; /* Socket=0 for dynamically assigned */
 block.csParam.atpOpenSkt.addr.net = 0; /* Zero internet fields */
 block.csParam.atpOpenSkt.addr.node = 0;
 block.csParam.atpOpenSkt.addr.skt = 0;
 skt = PBControl(&block, 0) ? 0 : block.csParam.atpOpenSkt.skt; / OPEN IT */
 return(block.ioResult); /* Return result */
}

/*
 * atpCloseSkt - Close an ATP responding socket.
 *
 * Returns I/O result code.
 */
short atpCloseSkt(skt)
unsigned char skt; /* Number of ATP socket to close */
{
 ATPBlock block;

 block.ioRefNum = atpRefNum; /* I/O is for .ATP driver */
 block.csCode = _atpCloseSkt; /* Function is "Close Resp. Socket" */
 block.csParam.atpCloseSkt.skt = skt; /* Fill in socket to close */
 return(PBControl(&block, 0)); /* DO IT and return result */
}

/*
 * SendATPResp - Send a single-packet ATP response.
 *
 * Currently, this routine only supports responses consisting of a single ATP packet.
 * The I/O parameter block is allocated statically because the I/O may be asynchronous,
 * thus the block must stay around after this routine exits. Same goes for the BDS,
 * which gets filled in here.
 */
SendATPResp(ATPSendResp, Dest_NetAddr, SourceSkt, Resp, RespSize,
 atpRespBDS, TransID, asynch)
ATPBlock *ATPSendResp; /* -> Statically allocated I/O parameter block */
AddrBlock *Dest_NetAddr; /* -> Destination network address for response */
byte SourceSkt; /* Local responding socket number */
char *Resp; /* -> response data */
unsigned short RespSize; /* Length in bytes of response data */
BDS *atpRespBDS; /* Statically allocated Buffer Descriptor Structure */
unsigned short TransID; /* ID of transaction being replied to */
short asynch; /* Boolean defining whether or not it is async. */
{
 unsigned short atpBDS();

 ATPSendResp->ioRefNum = atpRefNum; /* I/O is for .ATP driver */

```

```

ATPSendResp->csCode = _atpSendResponse; /* Function is "Send Response */
/*
 * Copy Requestor's net address into parameter block
 */
bcopy(&((ATPSendResp->csParam).atpSendResponse.addr), Dest_NetAddr,
 sizeof(AddrBlock));
(ATPSendResp->csParam).atpSendResponse.skt = SourceSkt; /* Responding Skt. */
(ATPSendResp->csParam).atpSendResponse.bdsPtr = atpRespBDS; /* Fill in -> BDS */
(ATPSendResp->csParam).atpSendResponse.bdsSize = 1; /* 1-packet resp. */
(ATPSendResp->csParam).atpSendResponse.numOfBufs = 1; /* (same) */
(ATPSendResp->csParam).atpSendResponse.flags = atpEOM; /* 1st has EOM */
(ATPSendResp->csParam).atpSendResponse.transID = TransID; /*
atpBDS(atpRespBDS, Resp, RespSize); /* Fill in the BDS */
if(asynch) /* Only 1/0 legal in PBControl */
 PBControl(ATPSendResp, 1); /* Do it asynchronously */
else
 PBControl(ATPSendResp, 0); /* Do it synchronously */
return(ATPSendResp->ioResult); /* Return result code */
}

/*
 * GetATPRequest - Get an ATP Request
 */
/* Issue an ATP "GetRequest". If a non-NULL completion routine address is supplied, issue the
 * request asynchronously.
 */
GetATPRequest(ATPGetReq, skt, ReqPkt, ReqPktSize, ioCompletion)
ATPBlock *ATPGetReq; /* Statically allocated I/O parameter block */
byte skt; /* Local responding socket on which to accept request */
byte *ReqPkt; /* -> buffer to get filled in with request data */
unsigned short ReqPktSize; /* Size of request buffer, bytes */
ProcPtr ioCompletion; /* -> entry point of completion routine or NULL */
{
 ATPGetReq->ioRefNum = atpRefNum; /* I/O is for .ATP driver */
 ATPGetReq->csCode = _atpGetRequest; /* Function is "Get Request" */
 (ATPGetReq->csParam).atpGetRequest.reqPtr = ReqPkt; /* Fill in buffer data */
 (ATPGetReq->csParam).atpGetRequest.reqSize = ReqPktSize;
 (ATPGetReq->csParam).atpGetRequest.skt = skt; /* Socket on which to receive req. */
 /*
 * Fill in completion routine address, and if non-NULL, issue asynch I/O otherwise
 * issue it synchronously.
 */
 if((ATPGetReq->ioCompletion = ioCompletion) != NULL)
 PBControl(ATPGetReq, 1);
 else
 PBControl(ATPGetReq, 0);
 return(ATPGetReq->ioResult); /* Return I/O result code */
}

/*
 * Miscellaneous Low-Level Routines
 */

/*
 * GetReqComplete
 */
/*
 * ATP Get Request Completion Routine simply posts an event with a unique event/message pair.
 * When the event is identified by the modal dialog filter procedure, a fortune is returned to the
 * requestor and another GetRequest is issued. Registers must be preserved because this routine
 * runs asynchronously, and may interrupt some other process, including a local fortune fetch.
 * It's written in assembler
 */

```

## GetReqComplete()

```
{
#asm
 Movem.L a0-a6/d0-d7,-(sp) ; Save registers
 Move.L _SavedA5,a5 ; Restore A5 context
 Move.W #MouseDown,-(sp) ; Event type = Mouse Down
 Pea ATPGetReq ; Event message is pointer to GetReq block
 _PostEvent ; Post it
 Movem.L (sp)+,a0-a6/d0-d7 ; Restore registers
#endasm
}
```

/\*

\* FilterGlue - Interface for ModalDialog Filter Proc. Callback

\*

\* This is a glue routine to get back into C. It is called back from ModalDialog() with

\* (naturally) Pascal-flavored arguments on the stack. See the C Workshop in MacTutor

\* Vol. 1, No. 7 (June, 1985) and Inside Macintosh for more information on ModalDialog

\* event filter procedures.

\*/

## FilterGlue()

```
{
#asm
;
; Inputs (Pascal callfrom ModalDialog):
; 4(SP) Address of word to fill in with item number
; 8(SP) Address of the event record
; 12(SP) Dialog (window) pointer
;
 Link a6,#0 ; no local automatics
 Movem.L d1-d7/a1-a4,-(sp) ; Save regs
 Move.L 8(a6),d0 ; D0 -> Item number word
 Move.L 12(a6),d1 ; D1 -> Event record
 Move.L 16(a6),d2 ; D2 -> Window record
 Jsr filter ; Call C for dirty work
 Movem.L (sp)+,d1-d7/a1-a4 ; Restore regs
 Unlk a6 ; "standard" Pascal routine exit
 Move.l (sp)+,a0 ; A0 -> return point
 Addq.L #6,sp ; Pop args
 Addq.L #6,sp
 Move.B d0,(sp) ; Copy C result to Pascal return
 Jmp (a0) ; Return
#endasm
}
```

/\*

\* filter - "Real" ModalDialog filter function

\*

\* If there is a "phoney" mouse down event (with message = address of our GetReq block)

\* then we have received a fortune request. In that case, send a fortune back to the requestor.

\*/

## filter(lp, ep, wp)

```
short *ip; /* -> Word to receive dialog item code */
EventRecord *ep; /* -> Event record */
WindowPtr wp; /* -> Window record for modal dialog */
{
 if(ep->what == mouseDown && ep->message == (long)&ATPGetReq)
 {
 SendFortune(); /* Send a responding fortune */
 ip = 0; / No item was hit */
 return(TRUE); /* Indicate that we handled the event */
 }
}
```

```

 }
 return(FALSE); /* We didn't handle this event */
}

/*
 * atpBDS- Setup an ATP Buffer Descriptor Structure.
 *
 * Given a buffer-load of data to send, fill in a BDS which describes the buffer as possibly
 * several ATP packets. Return the number of ATP packets required to send the data as
 * a response.
 */
unsigned short atpBDS(bds, buf, size)
BDS *bds; /* -> Buffer Descriptor Structure to fill in */
unsigned char *buf; /* -> Response data buffer */
short size; /* Size (bytes) of response data */
{
 unsigned short i; /* Counts number of segments */

 i = 0;
 while(size > 0) /* Loop till enough packets set up */
 {
 bds->buffSize = (size > atpMaxData) ? atpMaxData : size; /* Size of this packet */
 bds->buffPtr = buf; /* Fill in -> to this segment/packet */
 buf += atpMaxData; /* Advance segment pointer */
 size -= atpMaxData; /* Count down segment size */
 bds++; /* Advance BDS pointer to next seg. */
 i++; /* Count a segment */
 }
 return(i); /* Return segment count */
}

```

```

/*
 * pstrcpy - Pascal String Copy routine.
 *
 * Returns number of bytes copied, including count byte.
 */
pstrcpy(d, s)
char *d; /* -> Destination string */
char *s; /* -> Source string */
{
 bcopy(d, s, *s + 1);
 return(*s + 1);
}

```

```

/*
 * bcopy - General purpose byte copy routine.
 */
bcopy(d, s, n)
char *d; /* Destination address */
char *s; /* Source address */
short n; /* Number of bytes to copy */
{
 #asm
 Move.L d0,a0 ; A0 -> Destination
 Move.L d1,a1 ; A1 -> Source
 Bra.S @1 ; Jump past to correct for dumb DBRA inst.
 @0 Move.B (a1)+,(a0)+ ; Copy a byte
 @1 Dbra.W d2,@0 ; Loop till done
 #endasm
}

```

```

/*
 * SaveA5
 */
SaveA5()
{
 #asm
 Lea _SavedA5,a0 ; A0 -> Save cell
 Move.l a5,(a0) ; Save A5
 Rts ; Return

 _SavedA5 Dc.l 0 ; Storage area for A5
 #endasm
}

```



# *Programmer's Forum*

## *C Glue Routines for Filter Procs*



*Van Kichline  
John Pence  
MacTutor Vol. 1 No. 10*

The Macintosh ROM is divided into two sections, the operating system and the programmer's toolbox. The programmer's toolbox comprises about two thirds of the ROM and is there to make it easier for the programmer to adhere to Apple's stringent but cohesive user interface guidelines. It provides easy to use routines for creating windows and text edit records, dealing with resources, conducting modal dialogs and alerts, and many, many more functions. Used with a little care, it makes your program look and perform like a commercial Macintosh product, and makes it easy and intuitive for users to use your program.

The toolbox helps the programmer do it right, but what if you want to do it just a little differently? Not many individuals would care to rewrite and debug the routines provided by the toolbox, but in many cases there are alternatives built in. Many toolbox routines include parameters for optional filter or action procedures, which can be used with a default value (usually NIL) or with a pointer to a procedure you supply. Some examples are filterProcs for SGetFile and ModalDialog, and actionProcs for controls.

A dialog filterProc is invoked by calling ModalDialog with a procPtr to your filter procedure. It changes the way ModalDialog responds to events that take place within its domain. The object of one filter we wrote was to capture keystrokes that occurred while the command key was down, format them, and display them in a rectangle in the ModalDialog box. The filter looked at keyDown events, checked the modifiers field, changed the itemHit to 0 so that a TextEdit box in the same dialog wouldn't know about the keystroke, and then did a little string fiddling. It didn't take long to write, but it took a while to get running!

The Programmer's Toolbox expects you to be a Pascal programmer, not a C programmer, and C passes arguments quite differently than Pascal. This means that the ROM will call your function, but the way it presents its data is incompatible with Mac C. Pascal passes its parameters on the stack, and Mac C passes its parameters in registers.

Can non-Pascal programmers use filters and actionProcs at all? Is there any solution? Is this the end?

There are two solutions, actually. Assembly language routines can be used for all procedures that are called by the ROM. Assembly language is easy to mix with C programs, and allows the programmer the flexibility to deal with data in any format in which it may be presented. There is nothing at all wrong with this solution, and Inside Mac provides much valuable information for the assembly programmer. Assembly code is tight, fast, and efficient. Assembly programming, however, requires a firm grasp of the instruction set of the

processor, and is more difficult and time consuming than programming in a medium level language like C.

Functions called by the ROM can be written in C with a little care, a little effort, and a little glue. The term "glue" refers to a few assembly language instructions that fasten your code and the ROM code together. A glue routine is a labeled set of instructions that interface a particular function to another, "incompatible" function. Glue routines are easy to implement, and once a glue routine for a particular case is developed, it can be readily copied to other routines of the same type with only the most trivial modifications. This allows the programmer to rapidly write the filter and action routines in C. Once debugged and tuned, the routines can be converted to assembly if required, but I haven't found a need to convert any yet.

Pascal calls FUNCTIONS and PROCEDURES by pushing its arguments onto the stack. If the routine being called is a FUNCTION (a Pascal routine that returns a result) a place for the result is cleared on the stack, which may be two or four bytes wide. Then the parameters for the routine, which may also be two or four bytes each, are pushed on the stack in the order which they are declared in the Pascal procedure's definition. In other words, if the procedure Meza is being called, and it's defined:

```
PROCEDURE Meza(Homos : food ; Gyros: food ; Pita : bread) ;
```

Then the arguments would be pushed in the order Homos, Gyros, and Pita. When they're retrieved, they'll be popped in the order Pita, Gyros, Homos. Be careful. Think backwards. Finally, the JSR instruction that calls the procedure places the four byte return address on top of the stack, covering the parameters.

Mac C functions pass the values of the first seven arguments, assuming there are more, in the data registers D0 through D6. Excess arguments are stored on the stack, but we won't deal with the complexities of excess arguments here. The prologue code for each function defined in Mac C actually takes the arguments out of the registers and stores them on the stack in a "stack frame," but we are free to ignore what takes place once the C function is invoked. What's of importance to the writer of a glue routine is taking the Pascal parameters off the stack and placing them in the appropriate registers while preserving the return address. For Pascal FUNCTIONS, the result must also be placed in the appropriate location on the stack.

Subscribe to the oldest Mac Technical Journal  
**MacTutor** - publishing since 1984

## The Stack

The Pacal Function MyFilter pushes 18 bytes onto the stack.

```
FUNCTION MyFilter (theDialog : DialogPtr; VAR theEvent : EventRecord;
 Var ItemHit : INTEGER) : BOOLEAN ;
```

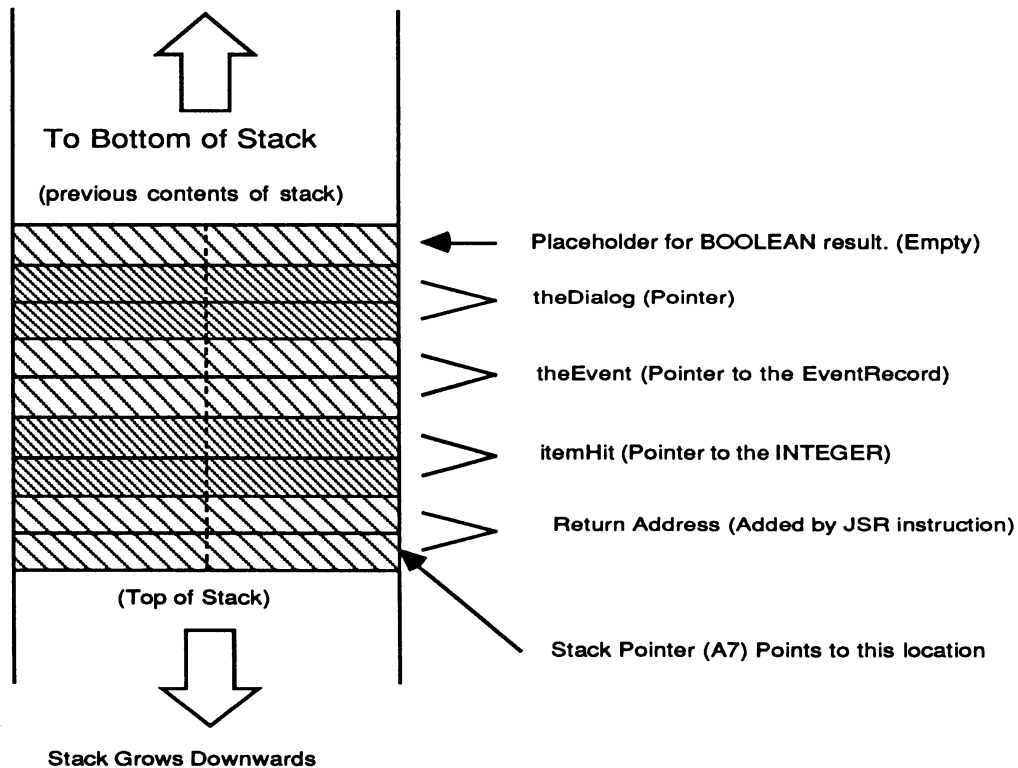


Fig. 1 Stack Operation

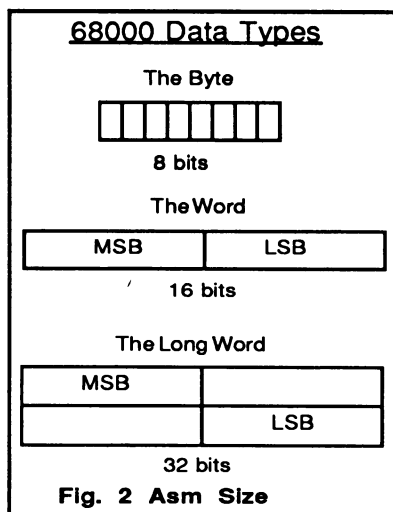
### Assembling the Solution

There are several ways to construct glue routines. The way I've presented here is applicable for routines requiring up to three parameters. Another way would be to "seal" the parameters in a stack frame and extract each parameter relative to A6. (See Robert Denny's column in MT 1, 7) I've selected the more direct approach because it's easier to describe, and is sufficient for all routines I've encountered except window and menu definition procedures, which are extremely complex. Our goal is to present enough information for the reader to construct his/her own glue routines without further aid, so we've selected the direct, or "brute force" method for presentation.

Assembly language may be included in Mac C source code by bracketing the lines with the terms #asm and #endasm. What goes in between is assembly source code that's exactly like MDS assembly source. Here's the skeleton of a Pascal to Mac C glue routine:

```
#asm
routineName:
MOVE.L (SP)+, A0 ; what you call the function
MOVE.X (SP)+, DX ; pop return address to A0
 ; save up to 3 params, of 2
 ; or 4 bytes.
MOVEM.L A0, -(SP) ; return address on top of
 ; stack
MOVEM.L A3-A4/D3-D7, -(SP) ; save the registers
JSR myFunctionInC ; execute the C code
MOVEM.L (SP)+, A3-A4/D3-D7 ; Restore registers.
MOVE.X X0, 4(SP) ; if it's a function, return a
 ; value
RTS
#endasm
```

The first line of the glue routine is the label, "routineName." This would be replaced with a unique function name in your application. The toolbox routine calls this label, not myFunctionInC. If the C function is called by the ROM instead of the glue routine, the system will crash. The label is declared as a C function at the beginning of the file. For example, if this were to be a ModalDialog filterProc, I would declare it in advance:



```
short routineName(); /* type short : returns Pascal
BOOLEAN */
```

Thereafter, the term "routineName" represents a pointer to the function. To use it as a function for a particular modal filter, you'd use:

```
do{
 ModalDialog(routineName, &itemHit) ;
 switch(itemHit)
 {
 case QUIT: (code) break
 case CANCEL: (code) break ;
```

```
 case CRASH: (code) break ;
 default: SysBeep(8) ;
 }
} while TRUE ;
```

Thus, the label of the glue routine is treated exactly as if it were the name of the C function it calls. The actual C function is referenced by nobody but the glue routine.

The second item in the glue routine skeleton pops the top four bytes off the stack and puts them in A0 for temporary storage. This is the return address of the routine calling our filter, pushed on the stack by a JSR in the ROM. In this case, it is ModalDialog who called, and the return address is the only way back to it.

Next is the part that does the actual gluing, and varies for different usages. Parameters, being two or four bytes in length, are popped off the stack and stored in data registers. (See the illustration "Data Configurations.") This data, remember, was pushed there by the toolbox routine before calling us and comprises the parameters our C function needs in its registers. If the data is two bytes in length MOVE.W is used, and if the data is four bytes long MOVE.L is used in place of MOVE.X. See the illustration of the stack at entry to the glue routine.

After the parameters are moved into the registers the return address, which we'd stored in A0, is placed back on top of the stack.

### C and Pascal Data Types

|                                                                                           | Pascal Type                               | C Type                  | Range                            |
|-------------------------------------------------------------------------------------------|-------------------------------------------|-------------------------|----------------------------------|
|                                                                                           | INTEGER                                   | short                   | (-32768 through 32768)           |
|                                                                                           | LONGINT                                   | int                     | (-2147483648 through 2147483647) |
| Void                                                                                      | BOOLEAN                                   | short                   | (True or False)                  |
| Result in Bit 0 of high order byte:<br>1 = True, 0 = False (ie \$0100=true, \$0000=false) |                                           |                         |                                  |
| Void                                                                                      | CHAR                                      | short                   | (0 through 255)                  |
| Extended ASCII code<br>in low order byte                                                  |                                           |                         |                                  |
|                                                                                           | Ptr, Handle, String,<br>& VAR parameters. | long, Ptr,<br>or Handle | (Address of parameter)           |

**Fig. 3 C versus Pascal Data Types**

Next, the register set is saved. The MoveMultiple instruction saves all the registers desired in a single line. Then, the JSR instruction to the private name myFunctionInC executes the real code.

After the last parameter is moved to the appropriate data register, the stack pointer (A7) points at the place holder for the FUNCTION result if there is one, or else to "unknown territory," or other essential data that remains on the stack and must be preserved. Next, the return address is placed back on top of the stack (four bytes) followed by the registers (28 bytes.) When the JSR myFunctionInC instruction is executed, it pushes a return address to the instruction following the JSR onto the stack and puts the address of myFunctionInC in the program counter. The data the C code needs is in the registers. The C function doesn't disturb anything on the stack except the return address on the very top, which it uses to return to the glue routine with an RTS.

Once the C function returns to the glue routine that called it, the registers that the glue routine saved are restored, popping them from the stack. Directly under those registers is the return address of the caller

that we were so careful to preserve earlier. If our C code was emulating a Pascal FUNCTION, the place holder for the result is directly under the return address. We must place the result of our function in this location. Mac C returns results that are values in D0, and results that are pointers in A0. So a BOOLEAN result would be in D0, and two bytes long. In this case, the last instruction before the RTS would be MOVE.W D0, 4(SP). It's always 4(SP), because the return address always four bytes long, but the instruction may move a word or a long word from D0, or a long word from A0, depending on the data type of the result. Don't put a result there if it's not required! You'll mash irreplaceable data and crash. Now a simple JSR propels us back into the ROM and the inner sanctums of ModalDialog.

### Concrete Glue

Here's a concrete example. The toolbox routine TrackControl can use a pointer to an actionProc as a parameter. This actionProc represents a continuous action to be performed while the control is being tracked. Scroll bars require an actionProc in order to make the arrows and paging parts work. First, the label is declared globally.

```
void trackScroll() ;
```

The function's declared as void because it's used as a Pascal PROCEDURE, and returns no result. Then, when a mousedown occurs in a scroll bar, the application finds the controls handle and calls:

```
if (TestControl(controlHand, &theEvent->where) == inThumb)
 TrackControl(controlHand, &theEvent->where, NIL) ;
else
 TrackControl(controlHand, &theEvent->where, trackScroll) ;
```

Note that TrackControl doesn't need an actionProc for the thumb (moving box part of the scroller), so why rewrite one?

If the else branch is taken, our glue routine is called by the ROM. An action proc for an indicator like a scroll bar receives two parameters; a ControlHandle and a short representing the partCode of the control that was activated. No result is returned. Thus the glue routine goes:

```
#asm
trackScroll:
MOVE.L (SP)+, A0 ; temp storage for return addr
MOVE.W (SP)+, D1 ; partCode goes in D1, 2 bytes
MOVE.L (SP)+, D0 ; controlHandle goes in D0, 4 bytes
MOVE.L A0, -(SP) ; push return address
MOVEM.L A3-A4/D3-D7, -(SP) ; save regs
JSR Cscroll ; do the function written in C
MOVEM.L (SP)+, A3-A4/D3-D7 ; restore regs
RTS ; no result. go back to TrackControl
#endasm
```

The labels trackScroll and Cscroll are specific to an implementation, while the rest is constant from one actionProc to another. The size of the parameters determines

which MOVE instructions to use. D1 is loaded first, then D0, because they were pushed onto the stack in order, and are popped off in reverse.

The glue routine may be contained entirely within the function called by it. This makes cutting and pasting the routine to another application easier. A complete, albeit simple example of a scrolling actionProc in C might be:

```
void deadCscrolls (theControl, partCode)
ControlHandle theControl ;
short partCode ;
{
 short amount, startVal, up ;

 if (!partCode)
 return ;
 startVal = GetCtlValue(theControl) ;
 up = (partCode == inUpButton ||
 partCode == inPageUp) ? TRUE : FALSE ;

 if ((up && (startVal > GetCtlMin(theControl))) ||
 (!up && (startVal < GetCtlMax(theControl))))
 {
 amount = (up) ? -1 : 1 ;
 SetCtlValue(theControl, startVal + amount) ;
 }
 return ;

 /* the Glue routine */
}
```

```
#asm
trackScroll: ; TrackControl calls trackScroll, not
; deadCscrolls!
MOVE.L (SP)+, A0 ; save the return address
MOVE.W (SP)+, D1 ; partCode to D1
MOVE.L (SP)+, D0 ; theControl to D0
MOVE.L A0, -(SP) ; return address goes here
MOVEM.L A3-A4/D3-D7, -(SP) ; save the registers
JSR deadCscrolls ; the scroll actionproc - C
MOVEM.L (SP)+, A3-A4/D3-D7 ; restore the registers
RTS ; there's nothing to return
#endasm
}
```

Note that every actionProc of this type uses the exact same glue routine. It may take a little while to work out the first time, but the effort doesn't need to be repeated. A small collection is all you need.

Glue routines can be used to rapidly implement toolbox modifying procedures and functions in C. Such implementation allows the programmer to write filters and actionProcs readily, and simplifies debugging and maintaining them. Such routines can be converted entirely to assembly after testing for greater efficiency, or may be left as is with little or no difference in performance.

Author's note: The techniques described here may or may not apply to other C compilers. We'd be interested in hearing.

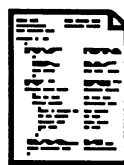




# C Workshop

## Programmer's Guide to Networking

C



Robert B. Denny  
Timothy T. Coad  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 11

### What Apple Doesn't Tell You!

It is deceptively easy to get started with AppleTalk. But once you get into non-trivial applications, things get difficult rather quickly. The drivers are poorly documented, there are a number of "features" and limitations that don't come to light right away, and there are a few bugs in the ROM that show up when using AppleTalk.

When I set about to write this article, I was completely surprised at the amount of information that I had digested over the last six months of working with AppleTalk. Back then, Tim Coad, Mike Schuster and I were given a set of pre-release drivers and a copy of Inside AppleTalk which had no documentation on the drivers at all.

We were desperate. We disassembled and commented the drivers. It started out as a "quickie" project that quickly became a time-burning obsession. We learned a lot about the internals of AppleTalk, and finally completed the project after about a month.

We then set about writing some "practice" applications and desk accessories. We all spent a good deal of time disassembling and tracing in the ROM. There are bugs in the current Mac ROM that have to do with handling asynchronous I/O and completion routines. The annoying part was that when we called Apple about the bugs, they said "Oh yes, we know about that one and it's fixed in the new ROMS." *I wonder why no mention was made in the software supplements?* In fact, I have yet to see any information from Apple about known bugs.

Anyway, the information we had gleaned from disassembling the AppleTalk drivers turned out to be of immense value in developing applications. There are many subtleties in using the MPP and ATP driver services, particularly in asynchronous I/O and completion routines.

I will try to share some of this information with you. To whet your appetite, Figure 1 shows the layout of the various components of the AppleTalk Manager (drivers and resources). The arrows are meant to show pointer relationships.

As usual for this column, the content is geared to the advanced programmer who is familiar with the Mac I/O conventions and services of the Device Manager. If you plan to do any development using AppleTalk, you'll need Inside AppleTalk, available from Apple. At a minimum, you'll need the AppleTalk Manager Programmer's Guide, furnished with the May/June "software supplement" for developers. Earlier versions contain many errors.

### Basic Concepts: Requestors and Responders

Before diving in, let's look at the sequence of events of a typical requestor and responder pair using ATP and NBP. We'll use the dial-a-fortune example described last month. The server accepts ATP requests and responds with a fortune cookie message.

First, the responder (dial-a-fortune *server*) opens an ATP responding socket, letting ATP assign a random (dynamic) socket number. Then it registers itself using NBP as an object of type "Dial-A-Fortune" with some arbitrary name field. At this point, the server can be located by name using NBP. Finally, the server issues an ATP "getRequest", making it ready to receive incoming fortune requests.

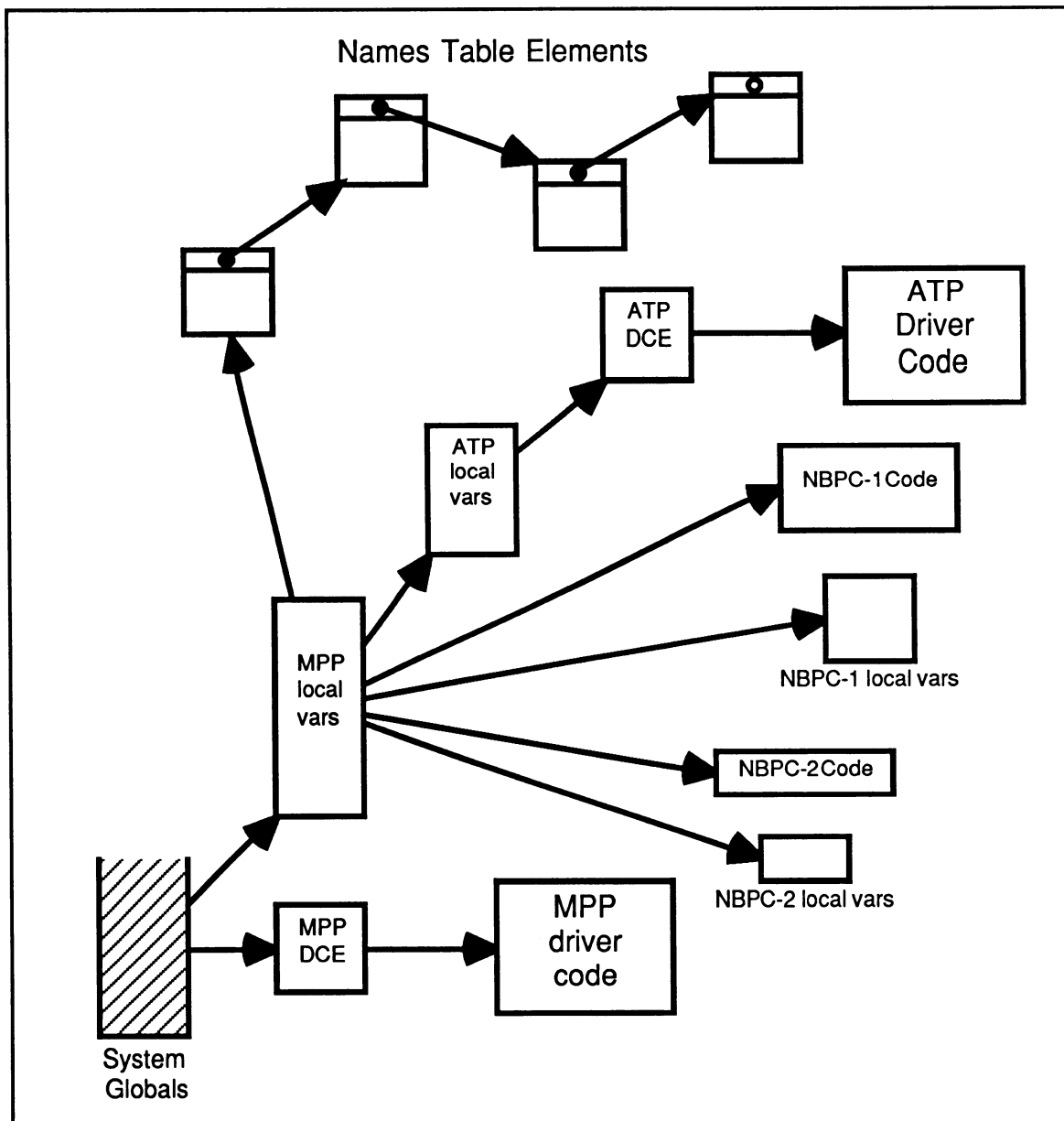
The requestor (client) must locate a server by name, since socket numbers are normally dynamically assigned. So it asks NBP to "Tell me the internet addresses of all objects of type 'Dial-A-Server'." Then it picks out a server and uses the corresponding internet address to issue a request. Note that the requestor does not "open" a socket with ATP. Instead, ATP opens a new socket automatically every time a request is issued, and closes it when the response is received or an error occurs. The requestor client need not be concerned with socket numbers.

The server responds to the request with a fortune cookie message. Then it issues another getRequest, making it ready for the next incoming request. Meanwhile the requestor receives the cookie response and displays it on the screen, or whatever. That's all there is to it.

### AppleTalk Data Structures In C

Last month's C Workshop covered the overall architecture of the AppleTalk network and presented the "dial-a-fortune" server application written (mostly) in C. There was not enough room to publish the file "ATALK.H", which contains the structure and symbol definitions needed to compile the example.

Much can be learned from studying the data structures involved when using the services of the AppleTalk drivers MPP and ATP. Therefore, we'll discuss some practical aspects of using AppleTalk I/O services while displaying the relevant C structure definitions. We shall limit the discussion to AppleTalk Transaction Protocol (ATP) and Name Binding Protocol (NBP). Most applications need only these services. Other parts of the AppleTalk manager are useful only in rare situations.



### Common Data Structures

```

#define byte unsigned char
#define word unsigned short
#define longword unsigned long

```

First, we define some new type names, "byte", "word" and "longword", which simply make the other structure definitions more easily readable.

```

/* AppleTalk Network Address Block */
typedef struct
{
 word net; /* Network number */
 byte node; /* Node number */
 byte skt; /* Socket number */
}

```

```

} addrBlock;

```

The Address Block contains the complete specification of an *internetwork address*. The network number is used for routing; once a packet gets to the destination net all nodes on that net can "hear" it, and the proper one picks it up and delivers it to the indicated socket in that node. The Address Block is used throughout for addressing network visible entities.

```

/* Unit and Reference Numbers */
#define mppUnitNum 9
#define atpUnitNum 10
#define mppRefNum -(mppUnitNum+1)
#define atpRefNum -(atpUnitNum+1)

```

Using Name Binding Protocol requires that you issue I/O requests to the 'MPP' device. Likewise, using AppleTalk Transaction Protocol requires I/O to the 'ATP' device. Their Device Manager *unit* and *reference* numbers are fixed as defined below:

### Definitions for Name Binding Protocol

```
/* Entity Name Tuple */
typedef struct
{
 addrBlock addr; /* Internet address */
 byte enumerator; /* Enumerator */
 byte entity[33]; /* Entity Name */
} NBPTuple;
```

The Name Binding Protocol provides the facilities for registering a *network-visible entity* (net, node and socket, the internet address) by name, and for locating registered entities by name and getting their net address. Registration assigns an entity name to a given internet address, and lookup returns one or more internet addresses given an entity name specification.

The entity name consists of three fields: the *name*, the *type* and the *zone*. Each field is a counted (Pascal) string up to 32 bytes in length. The name and type fields in a lookup request may contain an equal sign ("=") to indicate all names or types. The zone field always contains an asterisk ("\*"), meaning "this zone". Zone support will be implemented in the future. The individual fields are Pascal strings, and the delimiters are placed in between and are not counted. A complete entity name consists of those three fields separated by delimiters as shown below:

*name : type @\* (zone is always "\*")*

Entity names and their corresponding internet addresses form a pair called a *tuple*. When registering a new entity, you supply a tuple with the nbpRegister I/O request. When doing an nbpLookup, tuples are sent to you from other systems on the internet which have entities registered that match your lookup specification. The tuple structure shown below contains an "enumerator" field, which is used internally by the NBP software to speed up filtering of duplicate lookup responses from other nodes. You need not be concerned about this.

```
/* Names Table Entity Queue Element */
typedef struct _NBPElem
{
 struct _NBPElem *link; /* Next element */
 NBPTuple tuple; /* Entity tuple */
} NBPElem;
```

NBP keeps the names registered on its local node in a linked list. The tuple you supply when registering a name must be preceded by a longword that NBP will use as a list link. Together, the tuple and link longword are called a *Names*

*Table Queue Element*. The list doesn't really function as a queue; there is no implied ordering in the list. However, the Mac documentation frequently refers to order-free linked lists as queues, so we will follow Apple's conventions.

```
/* Retry Specification */
typedef struct
{
 byte interval; /* Retry interval, ticks/8 */
 byte count; /* Retry count */
} Retry;
```

NBP allows specification of retry interval and count on their I/O requests. The retry interval is specified in 8-tick units. Setting these values requires more careful thought than you might suppose at first.

If a remote node is too busy, it might not see a particular NBP lookup request (which may be a "real" request or a "verification" request made when checking the uniqueness of a name during registration). Therefore each "lookup" must really be a sequence of broadcast lookup requests, in the hope that everyone out there will see at least one of them and reply as required.

If you specify too few retries, or if the retries are too close together in time, some stations may miss the activity. This can result in your not finding something, or worse, registering a duplicate name. If you try too many times, it clogs the network with unnecessary traffic. If the retries span too long a time, your user will get sick of waiting for the lookup process to complete. This is the delay you get when the message "Looking for LaserWriter" appears on the screen during LaserWriter printing.

Name binding services are handled by the MPP driver, part of the AppleTalk Manager package. All NBP requests are mapped as *control* calls to the driver. The definition of the I/O parameter block used for MPP calls is shown below.

### NOTE

Never alter the contents of a parameter block used with AppleTalk until the corresponding I/O request completes. You must use a separate parameter block for each concurrent outstanding AppleTalk I/O request.

```
typedef struct _MPPBlock
{
 struct _MPBlock *ioLink; /* List link */
 short ioType; /* Next 3 used by Device Mgr */
 short ioTrap;
 Ptr ioCmdAddr;
 ProcPtr ioCompletion; /* -> Completion routine */
 short ioResult; /* Result code from req. */
 char *ioFileName; /* Unused */
 short ioVRefNum; /* Unused */
 short ioRefNum; /* Driver RefNum (-10) */
 short csCode; /* Op-code (see below) */
 MPPCsParam csParam; /* (see below) */
}
```

```
} MPPBlock;
```

```
/* NBP csCodes */
#define _nbpLoad 249 /* Load NBP */
#define _nbpConfirm 250 /* Confirm name */
#define _nbpLookup 251 /* Lookup name */
#define _nbpRemove 252 /* Remove name */
#define _nbpRegister 253 /* Register name */
#define _nbpKill 254 /* Kill NBP req */
#define _nbpUnload 255 /* Unload NBP */
```

The structure is typical for a control call as documented in the Device Manager section of *Inside Macintosh*. The csCode field selects which NBP service is being requested. The csParam field is a union of function-dependent structures relating to the particular service. This union is described later.

The \_nbpLoad and \_nbpUnload functions are normally not needed, as the AppleTalk drivers are loaded during startup on a 512K Mac. I haven't tried to do AppleTalk development on a 128K Mac, but I suspect it would be frustrating.

The csParam "field" is really a function-dependent continuation of the parameter block. In C, it's easiest to define this as a union, with each member struct defining the parameter layout for the corresponding driver service. It is not used with the nbpKill function, which simply kills any outstanding NBP I/O.

```
/* MPP Control Block */
typedef union
{
 struct /* For "Register Name" (csCode=253) */
 {
 Retry retry; /* Retry int. and count */
 NBPElem *elem; /* Pointer to entity spec */
 byte verify; /* Whether to verify */
 byte unused;
 } nbpRegister;
```

The first struct is used with csCode = \_nbpRegister. You supply the retry interval and count. This determines how hard NBP tries to check for a duplicate name. A good starting point is 16 ticks (retry interval = 16/8 = 2) and 20 retries. This means it will take 5.3 seconds to register a name. Hopefully, registering a name is an infrequent operation. What you don't want is a duplicate name, so it's worth it to try over and over again to be sure your name isn't in use.

You also supply the entity name table queue element, a tuple with a link field as described previously. Finally, you may disable the name uniqueness check. If you know that the name you want to register is unique, you can bypass the check and save time. Not recommended.

```
struct /* For Remove Name (csCode = 252) */
{
 word unused;
 Entity *entity; /* Pointer to entity spec */
} nbpRemove;
```

To remove a name, supply the entity specification.

Nothing else is needed.

```
struct /* For Lookup (csCode = 251) */
{
 Retry retry; /* Retry int. and count */
 Entity *entity; /* Pointer to entity spec */
 byte *retBuffPtr; /* Pointer to resp. buffer */
 word retBuffSize; /* Size of resp. buffer */
 word maxToGet; /* Max responses to get */
 word numGotten; /* No. actually gotten */
} nbpLookup;
```

The lookup is the most complex NBP operation. You supply the retry info, an entity specification, a place to put the tuples that you get back, and the maximum number of tuples that place can hold. When you issue the lookup request, NBP starts yelling all over the network "Hey, anyone out there have any names matching this description?" The retry information you supply governs the number of yells and the elapsed time between them.

If the entity specification contains any wildcard fields (name and/or type is "="), you may get back many tuples. Suppose you wanted to find all of the objects of type "Phone". You would supply an entity string of:

```
=:Phone@*
```

or "all phones" (in "this zone", but that's not really supported). NBP would then put into your buffer all of the tuples it got back from its repeated requests. Each unique tuple is entered only once. Then it fills in the "numGotten" field with the number of unique tuples received. The tuples are packed end-to-end in the buffer.

```
struct /* For Confirm (csCode = 250) */
{
 Retry retry; /* Retry int. and count */
 Entity *entity; /* Pointer to entity spec */
 addrBlock addr; /* addrBlock to confirm */
 byte newSkt; /* Current socket no. */
 byte unused;
} nbpConfirm;
} MPPCsParam; /* End of typedef union */
```

As we have seen, lookup requests can take an annoying amount of time to complete. NBP provides a way to "confirm" the internet address of a named entity that may have been looked up in the past. This makes it possible for an application to keep a "black book" of names and internet addresses locally, to minimize the need for NBP lookups.

This time, we supply the usual retry info and specific entity spec (no wildcards) plus our "guess" at the corresponding internet address. If our target object is still registered at that address, we'll get back a confirmation. NBP will stop as soon as it gets a single confirmation, and only the target node will respond, so this a lot faster.

Note the newSkt field in the confirm parameters. If the named object has changed socket numbers, but is otherwise at the same internet address, the confirm will still locate it. Why is this important? Most named objects on the network will be

assigned *dynamic* socket numbers. If the target system is rebooted, for example, the process which owns the socket in question may have been assigned a new and different socket number. Since dynamic socket numbers are randomly assigned each time a socket is opened, it is important to be able to confirm with a change in socket number. *Always use the returned socket number.*

## NBP - Hints & Inside Information

NBP has a few quirks and gotchas that aren't mentioned in the Apple documentation. First, NBP is not actually implemented "in" the MPP driver. Rather, its code is contained in two resources of type NBPC. Resource NBPC-1 contains the code for NBP initialization, and for looking up and confirming names on other nodes. NBPC-2 contains the code implementing the name server functions, used only if names are registered on the local node. Refer to Figure 1.

Presumably, this splitting was done to make the most out of the limited memory available on the 128K Mac. On that system, you must manually "load NBP" via an MPP driver call, and the NBPC-1 resource goes into the application heap. Then if you register a name, the NBPC-1 code reads in the NBPC-2 code, again into the application heap. When your application exits to the Finder, the NBPC resources are disposed, and all registered names are lost. On the 512K Mac, both resources are read into the system heap by the MPP open routine, and they remain there permanently.

When an application registers a name, it supplies the name table queue element. Normally, this is located in the application heap. The registration process simply links the queue element into the list of registered names. The application should take care to explicitly remove any registered names before exiting to the Finder.

If it doesn't, an interesting "safety net" process takes place. The MPP driver has the "need goodbye" flag set in its DCE flags. When an application does an `_ExitToShell`, the Memory Manager is called to re-initialize the application heap in preparation for loading and starting the Finder (or whatever). During this process, each driver which has need goodbye set receives a control request with a negative `csCode`. The driver (or desk accessory) is thus given a chance to clean up after the application.

In the case of registered names, this is very important. It is possible that a server driver may have registered a name permanently, in the system heap. The linked list of names table queue elements would be corrupted if the application heap was initialized when it contained names table entries.

When the MPP driver receives its need goodbye call, it checks to see if the NBPC-2 resource is loaded. If so (always on 512K), it calls a special routine in the NBPC-2 code which unlinks any names table entries that it finds in the application heap. It does not touch names in the system heap. This action protects the integrity of the names table linked list.

This has important implications for desk accessories which have registered names. A desk accessory is a driver, and if it has its need goodbye flag set, it will get called at application

exit. If it has a registered name, the desk accessory may explicitly call NBP to remove it as part of its clean-up operation.

What if the MPP driver gets called for need goodbye before the desk accessory? By the time the desk accessory gets its need goodbye call, the names in the application heap have already been removed by the MPP cleanup call to NBPC-2. When the desk accessory issues its `nbpRemove` call, it will return with an error because the name has already been removed. The desk accessory should check for this error and ignore it.

Another thing that isn't documented is that *only one NBP lookup request can be active at any time*. Each lookup request is sent with a "transaction ID", and the incoming replies to that request must carry that "TID". This mechanism was presumably meant to allow multiple concurrent NBP lookups. Using the TID in the reply, the NBP can match the reply to the appropriate lookup request.

In the current implementation of NBP, the TID is kept in a single memory location in NBP's private variables. When NBP starts processing a new lookup request, it creates a new TID and jams it into the single memory cell. This instantly invalidates any incoming replies from previous lookup requests, since the TID's will no longer match.

One last thing to keep in mind is that names registered on your own node are invisible to your NBP lookups. The only way to find out what's registered on your own node is to locate the names table and search it directly. Be careful not to alter any of the information in the names table.

At location `$2D8` is a pointer to MPP's private variables. At offset `$90` into this area is the pointer to the first names table entry, the head of the list. See Figure 1 for an overview. This is not documented in *Inside AppleTalk*, nor does the offset appear in any of Apple's "equates" files, so use this for experimenting only.

## Definitions for AppleTalk Transaction Protocol

Due to space limitations, this section assumes that you have access to the *AppleTalk Manager Programmer's Guide*, and have a reasonable understanding of ATP's theory of operation, including the "exactly once" concept.

ATP provides an error-free request/response type service. The request can carry up to 578 bytes of data, and the response can consist of up to 8 blocks of up to 578 data bytes each. The send and receive ATP I/O functions require that each of these data blocks be described by a *buffer descriptor structure* (BDS). The meanings of some fields vary depending on whether the BDS is being used with a send or a receive request.

```
/* Buffer Data Structure (S=Send, R=Receive) */
typedef struct
{
 word buffSize; /* S: data len R: buf len */
 byte *buffPtr; /* S: data addr R: buf addr */
 word dataSize; /* S: used int. R: data len */
}
```

```
longword userData; /* 4 user bytes */
} BDS;
```

```
typedef struct
{
 BDS bds[8];
} BDSType;
```

The userData field deserves special mention. Suppose you want to build a server that delivers blocks of information that exceed 578 bytes in length. No problem, the ATP response from the server may contain up to 8 blocks of that size. Simply set up buffer descriptors to point to contiguous chunks of the response data in the responder application's buffer. The requestor application can set up receiving descriptors which point to chunks of a contiguous buffer. ATP will then reassemble the chunks at the receiving end, leaving the large data block ready to use.

But suppose the response needs some sort of "tag" or other auxiliary information. If you put it into the data, it would make the just described automatic fragmentation and reassembly technique difficult to use. That's where the userData longwords come in.

For example, the AppleTalk Printer Access Protocol (PAP) uses ATP for error and flow control, while maintaining the notion of "connections" between printing Macs and a LaserWriter. PAP uses the userBytes field to carry the "connection ID" for a particular request/response pair, the PAP message type (op-code), and an optional EOF marker.

Each BDS describes a single packet of data to be sent or received by ATP. Responses may carry up to eight such packets. When issuing a multi-packet response, the individual buffer descriptors are set up as an array of descriptors as defined above.

```
typedef struct _ATPBlock
{
 struct __ATPBlock *ioLink; /* I/O Queue Link */
 short ioType; /* Next 3 used by device mgr */
 short ioTrap;
 Ptr ioCmdAddr;
 ProcPtr ioCompletion; /* -> Completion routine */
 short ioResult; /* I/O result code */
 longword userData; /* 4 user bytes */
 short transID; /* Transaction ID */
 short ioRefNum; /* ATP refNum (-11) */
 short csCode; /* Op-Code (see below) */
 ATPCsParam csParam; /* Function-dependent */
} ATPBlock;
```

```
/* ATP csCodes */
#define _atpRelRspCB 249 /* Release RspCB */
#define _atpCloseSkt 250 /* Close socket */
#define _atpAddResponse 251 /* Add response */
#define _atpSendResponse 252 /* Send response */
#define _atpGetRequest 253 /* Get request */
#define _atpOpenSkt 254 /* Open socket */
#define _atpSendRequest 255 /* Send request */
#define _atpRelTCB 256 /* Release TCB */
```

ATP services are handled by the ATP driver, part of the AppleTalk Manager package. All ATP requests are mapped as *control* calls to the driver. The definition of the I/O parameter block used for ATP calls is shown below. Note that the AppleTalk Manager Programmer's Guide is wrong about the reqTID field offset for the sendRequest function.

The structure is typical for a control call as documented in the Device Manager section of Inside Macintosh. The csCode field selects which ATP service is being requested. The csParam field is a union of function-dependent structures relating to the particular service. This union is described later.

```
/* ATP Control Block */
typedef union
{
 struct /* for OpenSkt (csCode = 254) */
 {
 byte skt; /* Socket number or 0 */
 byte unused;
 addrBlock addr; /* Request filter data */
 } atpOpenSkt;
```

The csParam "field" is really a function-dependent continuation of the parameter block. In C, it's easiest to define this as a union, with each member struct defining the parameter layout for the corresponding driver service.

The open responding socket function is used by server type applications to set up a network-visible entity which receives ATP requests and responds to them. If you supply a non-zero socket number, it will be used if possible. If the socket number is zero, ATP assigns a *dynamic* socket number. Normally, you should only use dynamic sockets. Statically assigned sockets are meant for special generic services that cannot use Name Binding Protocol for name-based references.

You may have ATP filter incoming requests by filling in the addrBlock with non-zero fields. If any of the addrBlock fields is non-zero, ATP will discard incoming requests that are not from the indicated socket, node and/or network.

```
struct /* for CloseSkt (csCode = 250) */
{
 byte skt; /* Responding socket to close */
 byte unused;
} atpCloseSkt;
```

To close an ATP responding socket, simply issue the close function and pass the socket number. If you have registered the socket using NBP, be sure to remove the name before closing the socket.

```
struct /* For SendRequest (csCode = 252) */
{
 byte bitMap; /* Error ctrl bitmap */
 byte flags; /* C/S flags */
 addrBlock addr; /* Resp's internet addr */
 word reqSize; /* Length of req. data */
 byte *reqPtr; /* -> Request data */
 BDS *bdsPtr; /* -> 1st response BDS */
}
```

```

byte numOfBufs; /* No. of resp. buffers */
byte timeOutVal; /* Timeout interval (sec.) */
byte numOfResps; /* No. resp blocks rec'd */
byte retryCount; /* Retry count */
} atpSendRequest;

```

Once you have determined the internet address of an ATP responder, you may make requests to its responding socket by issuing a `SendRequest`. The request data is not described by a BDS, instead you supply a pointer and length. The user data bytes are also part of the parameter block.

As with NBP, the retry interval and count is an important design consideration when using ATP. It is common for an ATP request to be missed by the responder, usually because the responder is busy servicing a previously received request. Some responders attempt to reduce their "blind window" by issuing a series of asynchronous `getRequests`. Nonetheless, it is still important for the requestor to make multiple attempts to get the request across.

Unlike NBP, an ATP request may complete before the retries are exhausted. If for some reason the server is unable to respond at all, the request will take the full time (retry interval times count) before completing with an error. Normally, you should try very hard to complete an ATP request, but if it is common for the server be unable to respond, you might not want to make your user wait too long to find out.

The I/O function completes when all blocks (up to 8) of the response have been received, or the retry count is exhausted. If you issue the request asynchronously, you can monitor its progress by watching the retry count and bitmap. The retry count is decremented with each retry, the bitmap indicates which blocks of the reply have already been correctly received. The error control bitmap is used during the progress of receiving the reply to tell which pieces of the reply have been received. *Be sure to clear the bitmap prior to issuing the `sendRequest`.*

The `atpXObit` flag is used to request the "exactly once" mode of transaction. This should be used when you want the action generated on the server by your request to be performed only once. Remember that your request must be tried several times. In practice, you'll probably use this mode most of the time.

```

struct
{
 byte skt; /* Socket to listen on */
 byte flags; /* Request flags */
 addrBlock addr; /* Reqstr internet addr */
 word reqSize; /* Length of reqst data */
 byte *reqPtr; /* -> Rqst data buffer */
 longword unused1;
 byte bitmap; /* Request bitmap */
 byte unused2;
 word transID; /* Transaction ID */
} atpGetRequest;

```

The `GetRequest` function is issued by the responder to

prepare for receiving a request. The socket on which to listen must have previously been set-up with an `OpenATPSkt`. Remember that responding sockets are "permanent" and requesting sockets live only as long as the request is in progress.

A lot of information is returned in the parameter block when the request completes. You get back the four `userData` bytes as a longword, the requestor's TID, `atpFlags` byte and bitmap, the internet address of the requestor and the actual length of the request data placed into your buffer. The bitmap is used to determine the number of blocks that the requestor expects in the response. This is required information for sending back the response (which may be sent back in pieces), as we shall see later.

Normally, requests can come from anywhere in the internet. Anyone who discovers your internet address can send you a request. You, as a responder, must be prepared to deal with "foreign" requests. It is possible that a request will have been meant for a different responder/server, and the requestor is using an out of date internet address (perhaps he should have done an NBP confirm?). The simplest solution is to simply discard the unknown requests. It's not too efficient, though, because the requestor will retry the request a number of times, wasting network and machine bandwidth. There is no way to "refuse" a request in ATP.

It may be possible to restrict the set of internet addresses from which requests will be accepted by putting non-zero values into the `addrBlock` field of the `getRequest` parameter block. This is documented in the "latest" `AppleTalk Manager Programmer's Guide`, but I personally have not tried it, and it's not in the code we disassembled last Winter. If it works, it could be a useful way to make a conversation private after an initial exchange of information

```

struct /* for sendResponse (csCode = 252) */
{
 byte skt; /* Response socket no. */
 byte flags; /* Response flags */
 addrBlock addr; /* Reqstr internet addr */
 word unused1;
 longword unused2;
 BDS *bdsPtr; /* -> Response BDS's */
 byte numOfBufs; /* # blocks in response */
 byte bdsSize;
 word transID; /* Transaction ID */
} atpSendResponse;

```

Once a server has received a request, it usually performs some action then sends back a response, using the `SendResponse` function. Normally, all blocks of the response will be sent back in a single `SendResponse` operation. It is possible to send a partial response, however, then use the `AddResponse` function described later to send the remaining blocks that make up the complete response.

If you are sending back the complete response with the `sendResponse`, you must indicate this by setting the EOM bit in the flags. Also, if the response is part of an "exactly once" transaction, you must set the XO bit in the flags. The latter

requirement is not documented; we found it to be necessary through inspection of our disassembled drivers. It may not be necessary in later versions of the drivers, but it can't hurt.

The socket number must match that of the socket on which the corresponding request was received. The internet address must be set to that of the requestor to whom the reply is being sent. The same goes for the TID, it must match that of the request.

The presence of both the numOfBufs and bdsSize fields can be confusing. bdsSize indicates the maximum number of blocks (up to 8) that may be sent back in the response, and numOfBufs indicates the number of blocks being returned in this sendResponse. More blocks may be added to the response using the addResponse function to be discussed next. Normally, though, you will set the two values to be equal and send the entire response back with the sendResponse.

```
struct /* for sendResponse (csCode = 252) */
{
 byte skt; /* Response socket no. */
 byte flags; /* Response flags */
 addrBlock addr; /* Reqstr internet addr */
 word unused1;
 longword unused2;
 BDS *bdsPtr; /* -> Response BDS's */
 byte numOfBufs; /* # blocks in response */
 byte bdsSize;
 word transID; /* Transaction ID */
} atpSendResponse;
```

Once in a while, a situation arises in which it is desirable to overlap the processes of preparing the response data and sending back the response. Suppose the request is to look up a series of records in a complex data base and return the records in the response. If the lookup process is time-consuming, you may want the responder to return a partial result in the sendResponse, which would be issued asynchronously, then complete the transaction by issuing addResponse calls as additional pieces of information become available. The EOM bit must be set in the flags byte of the last block.

Note that this function does not use a BDS. Rather, each addResponse can send a single block. For example, if the response is to be made up of 6 blocks and one is sent with the sendResponse, then five additional addResponse calls would be needed to complete the response.

There is a snag in this if the response is part of an XO transaction. The sendResponse does not complete until the requesting end sends a "release" indicating that it has received the complete response. If you send a partial response with the sendResponse, intending to finish with one or more addResponse calls, you'll get stuck unless the sendResponse is issued asynchronously. In this situation, it is also very important that any buffers supplied in addResponse calls not be touched until the sendResponse completes. The safest way to manage a multi-phase response is to use separate parameter blocks and buffers for each phase and leave everything alone until the sendResponse completes.

```
struct /* For RelRspCB (csCode = 249) */
{
 byte skt; /* Responding socket */
 byte unused1;
 addrBlock addr; /* Source of request */
 word unused2[6];
 word transID; /* TID of request */
} atpRelRspCB;
```

The "release response control block" function is used by the *responder* when it decides to "forget" about an exactly-once transaction before it receives the release from the requestor or it times out. It has the effect of cancelling an outstanding sendResponse. The requestor may never get the response. This function should be used only when a responder wants to abort a transaction.

```
struct /* for relTCB (csCode = 256) */
{
 word unused1;
 addrBlock addr; /* Rspndr internet addr */
 word unused2[6];
 word transID; /* Transaction ID */
} atpRelTCB;
} ATPCsParam;
```

If you want to kill an outstanding SendRequest, issue a relTCB. You must have the transaction ID (in the sendRequest parameter block). The reqTID field will contain the TID *only after ATP has assigned it*. You must wait for the tidValid bit in the flags to be set before you can trust the value in reqTID. This is documented in the AppleTalk Manager Programmer's Guide, but it's easy to miss.

#### NOTE

It is not necessary for the requesting client to manually send the XO release to the responder. The ATP driver does this automatically when it sees that the complete response has been received.

### ATP - Hints and Inside Information

There were a number of errors and omissions in early AppleTalk Manager Programmer's Guides, including the version in the Promotional Edition of Inside Macintosh (the "telephone book"). It is important that you get a version at least as recent as that shipped with the May/June 1985 Software Supplement.

The Apple documents make an almost invisible mention of a very important characteristic of ATP: *ATP completion routines are called with SCC interrupts disabled*. This has far-reaching consequences. The SCC is used for serial communications, Imagewriter printing, many brands of hard disks and most important of all, to control the mouse. If SCC interrupts are disabled, the mouse is frozen, and no serial port activity can take place.

Often an ATP responder will issue an asynchronous getRequest specifying a completion routine, then return to the system so that useful things can happen while the responder



waits for an incoming request. This sort of thing is a necessity if the responder is a desk accessory.

When the request arrives, the responder system's current activity is interrupted and the `getRequest` completion routine is called. If the completion tries to issue a synchronous `sendResponse`, the system will deadlock with a frozen mouse. The `sendResponse` won't complete until the SCC is turned back on, but the SCC won't be re-enabled until the completion routine returns, and it won't return until the `sendResponse` completes, etc. There's more ...

What if the responder has to look something up on disk before it can respond? If it has to use the floppy, the system becomes totally blind to anything on AppleTalk and the serial port for the duration of the floppy I/O. But what if the disk runs off of the serial port? Deadlock city with frozen mouse. The completion routine issues I/O to the disk which needs the SCC to function. Too bad.

Solutions to this dilemma have one thing in common. I/O should be deferred until the completion routine returns. It is important that *time spent in AppleTalk completion routines must be kept to a minimum.*

If the response requires little processing, it is possible issue the `sendResponse` asynchronously, then return from the completion routine. The SCC interrupts will be enabled and the `sendResponse` will proceed. **WARNING:** In the current ROM, there is a timing bug which will cause you trouble if you try to issue I/O in the completion routine. This is supposed to be fixed in the next release of the ROM. The problem is explained later in the article. For now, avoid doing any I/O during a completion routine or follow Mike Schuster's prescription for avoiding trouble.

If the responder is an application, the completion routine may post an event for the application to pick up the next time around its event loop. Then it can process the request and issue the `sendResponse` from the normal application state.

If the responder is a desk accessory, the completion routine can set a flag which is tested each time the desk accessory receives an `accRun` call from the system. If the flag is set, the desk accessory can process the request and issue the `sendResponse` asynchronously, then return to the system and the application.

There is one feature of the current ATP driver that is not documented in any of the programmer's guides, but is hinted at in Inside AppleTalk's ATP section. There is a facility where the responder may ask the requestor for a status report. The responder sets the `atpSTS` bit in the flags byte of a `sendResponse` or `addResponse`. When the requestor ATP receives a response with the STS bit set, it retransmits the request message with the current response bitmap. I haven't figured out why this is useful yet, but the support is there.

ATP is implemented in the '.ATP' driver (see Fig. 1) and is a client of the Datagram Delivery Protocol (DDP). ATP uses DDP as a "best efforts" routing and delivery service for its packets. There are a few things about DDP that you should know.

DDP can manage only twelve concurrent open sockets. This limit is imposed by the speed with which the 68000 can

search the socket table for a match during reception of an incoming packet. That's right, the 68000 is completely consumed during reception of an AppleTalk packet. It loops waiting for the SCC ready bit to set, then reads the byte, then decides what to do next, then loops waiting for the next byte to arrive, etc. It's the "decides what to do next" part that limits DDP's capacity.

As soon as the byte containing the socket number arrives, the DDP protocol handler has to search the socket table to locate the socket listener's entry point, then call the socket listener, which is responsible for controlling the reception of the rest of the packet. The developers presumably found that the largest socket table that could be reliably searched contained twelve entries for sockets.

When AppleTalk is loaded and started, one socket is immediately allocated to the routing information listener, who is responsible for keeping track of any internet routers on the local network. Later, if a name is registered on the local node, the "names information socket" is opened by the local NBP, making it possible for other nodes to locate the locally registered name(s).

So you might as well consider two sockets to be the property of routing and names information processes. This leaves ten sockets available for ATP.

Meanwhile, ATP has limits in its data structure allocations. Alert readers may have noticed the terms "transaction control block" (TCB) and "response control block" (respCB) used in the previous section. These are internal control structures used by the ATP requestor and responder, respectively, to maintain the status of an in-progress transaction.

ATP contains space for six TCB's, six open responding sockets, eight respCB's and 3 "I/O control blocks" (IOCB's). Each of these allocations imposes a limit on ATP's capacity. None of this is documented in the AppleTalk Manager Programmer's Guide, and believe me you can go crazy diagnosing errors without understanding these limits.

The limit of six TCB's means that a maximum of six concurrent ATP requests can be active. There can be no more than six responding sockets open at once. The respCB allocation means that there can be no more than eight concurrent XO responses in progress (there may be multiple XO responses going on a single responding socket).

Finally, the limit of three IOCB's means that there can be no more than three concurrent active DDP I/O requests. This deserves further explanation. DDP requests are active *only briefly*, when ATP needs to send a packet, and when it needs to open or close a DDP socket (which eventually becomes the ATP socket). This includes the following ATP operations:

|                          |                               |
|--------------------------|-------------------------------|
| <code>openATPSkt</code>  | <code>closeATPSkt</code>      |
| <code>sendRequest</code> | <code>sendResponse</code>     |
| <code>addResponse</code> | <code>relRspCB</code>         |
| <code>relTCB</code>      | <code>close ATP driver</code> |

If you try to close the ATP driver when there are not enough IOCB's available, you may be in trouble. The ATP driver's "close" routing *loops* waiting for a free IOCB, creating an

opportunity for a deadlock. Moral: clean up before closing the ATP driver. In fact, don't close it at all unless you're running on a 128K system.

An ATP `getRequest` does *not* use an IOCB because the reception of ATP packets is handled by the ATP socket listener, which is attached to the socket when the DDP socket is opened. No further DDP I/O is needed.

If you have a busy multi-tasking application which issues lots of asynchronous ATP requests, it is possible to run out of IOCB's. There is a specific error code for that case. Now you know what it means. Since IOCB exhaustion is a timing-related problem, one way to deal with the error is to have the requesting process sleep for a while then try the ATP operation again.

With all these limits in mind, don't lose track of DDP's total limit of 12 sockets. An ATP request uses a socket for the duration of the request. An open ATP responding socket obviously uses a DDP socket. An NBP lookup request uses a DDP socket for the duration of the request.

Finally, Mike Schuster found a serious bug in the ROM's file and device queue management code. The effect of this bug is that if you perform an I/O call within an SCC interrupt routine or within a VBL task, your Macintosh may bomb. When the following code is executed, `a0` contains a pointer to the I/O parameter block:

```
403924 lea $360,a1 ; a1 -> to file system
 ; queue header
403928 jsr *-$2E08 ; Enqueue pb (a0) on file
 ; system queue (a1)
40392C move sr,-(sp) ; save current status reg.
40392E ori #$300,sr ; dsab! SCC/VBL interrupts
403932 bset #0,$360 ; set file system busy bit
```

Suppose an SCC or VBL interrupt occurs during executing of the instruction at \$40392C. At this time, the parameter block has been enqueued on the file system queue, but the busy bit has not been set. If the interrupt handler performs an I/O trap, then that I/O as well as the one in the queue will be

completed before the interrupt handler returns. When it does return, the file system queue is empty, and the instruction at \$403932 sets the busy bit. Code following \$403932 then attempts to dequeue a parameter block and execute its command code, causing a bomb.

A similar sequence of code for device drivers begins at \$40118E. This code exhibits the same problem, except that the timing hole is two instructions wide. Thus, if the file system or a device driver queue is not empty and the busy bit is not set, you *must not* perform the I/O trap.

What is the solution? Simple. If the queue is not empty and the busy bit is not set, set the busy bit yourself, perform your async I/O trap, and reset the busy bit. Hence, your I/O is simply enqueued, and when you return, the interrupted ROM code processes the queued I/O calls correctly.

This bug has important implications to clients of the AppleTalk protocols, since (as we know) the `ioCompletion` calls are performed at SCC interrupt time. For example, if your `ioCompletion` routine for a ATP `GetRequest` call does a `SendResponse`, be sure to check the state of the ATP driver's I/O queue in its DCE. The busy bit in a DCE is bit 7 of the high order byte of the `dCtlQueue` word.

### Final Words

This is the longest article I have written for MacTutor. It's also the most disorganized. I don't know whether to apologize for the rambling style. There was so much information to present, I was almost forced to adopt a sort-of "stream of consciousness" approach, loosely organized around the data structures needed to compile the program in last month's column.

If you find this level of information useful and/or interesting, and would like to see more, please write to me, care of MacTutor, and let him know.





### Creating Tools and Mini-applications

A feature of the Macintosh that sets it apart from other computers is its ability to have **Desk Accessories**. Desk Accessories can be considered a 'mini-application' that can be executed while using another program, or while writing a Macintosh application program. Desk accessories are small programs that help you achieve a main goal, and are a part of the desktop concept that Apple has envisioned. In short, while creating programs or documents, there are tools we use intermittently, such as the Scrapbook or a Hex Calculator. This is reminiscent of activities before the advent of personal computers, when tools physically sat on the desk. Now the tools are more powerful electronic versions.

### Can Useful Desk Accessories be Written?

Desk accessories can be very powerful, enhancing the functionality of the Macintosh. My favorite example is **Click-On Worksheet** from T/Maker Graphics. Click-On Worksheet is a very powerful spreadsheet and chart program, that allows the creation of a small spreadsheet or chart while using a word processing or database program. Another is the **dCad Calculator** from Desktop, a scientific and programing calculator.

### Some Background Information

When writing that first desk accessory, there are two very important chapters in *Inside Macintosh* to familiarize yourself with: the **Desk Manager**, and the **Device Manager**. The sole purpose of the Desk Manager is to coordinate activities between the executing desk accessory and the application program. The Device Manager handles the I/O of information to and from the desk accessory.

When writing a desk accessory, the program content is considered a '**DRVr**' (driver) resource type from a Macintosh viewpoint. '**DRVr**' resources are kept in the System file and are read into memory by the Device Manager as needed. At this time the resource ID's of the DRVr's can range from 0 to 31, however remember that the first 11 are reserved by Apple. These reserved IDs are for things like the Disk, Sound, AppleTalk, and Printer drivers.

### Open, Close, Control, Prime, and Status

A '**DRVr**' routine is made up of 5 routines that dictate actions. They are **Open**, **Close**, **Control**, **Prime**, and

**Status**. The **Open** routine reads all the '**DRVr**' code and resources into memory, creates data structures and variables when the desk accessory is executed for the first time. The **Close** routine deletes and removes from memory structures and resources that the desk accessory used. The **Control** routine is the most important of all five routines. This routine is executed every time the procedure **SystemTask()** is called within the event loop of the application program currently running. Think of the procedure as giving control of the Mac over to the desk accessory, then when the desk accessory is closed, control is handed back to the main application program. The **Prime** and **Status** routines deal with the reading and writing of information to the '**DRVr**' and at this time are not important to us. One neat feature of Desk Accessories is the ability to attach and use resources such as **WIND**'s and **MENU**'s.

### Getting Started on the Desk Accessory

OK, let's get started on the Desk Accessory. This desk accessory has been written using 'C' as the language and most C compilers support the creating of desk accessories. Even if you don't program in C, most programs are easy to follow because of all the toolbox calls that programs use. The type of desk accessory that we will be writing is called a **Shell**. **Shells** allow a person to start from a point that works. Then, in a modular fashion, new routines and functions can be added, which allows you to track down errors and debug the program more easily. Once finished with this shell program, you will have a base to work from as you try out your own desk accessories.

At the beginning of the program listing the necessary header files have been included which represent predefined constants and variable structures. Next comes a procedure called **ACC()**. This is a MegaMax dependent item, since the Linker supports the creation of desk accessories. What is important are the arguments of the **ACC** procedure.

The **Control** routine can respond to certain device routines, which are determined by the argument **drvFlags**. Explanation of the flags can be found on pages 19 and 20 of the Device Manager Guide. How you determine the **drvFlags** is tersely explained in *Inside Macintosh*, so only after much thinking did the light come on! At the top of page 20 is a set of predefined constants such as **dNeedTime**. Looking at the equates, you see that the constants range from 0 to 6. These numbers represent which bit in the high word byte of the **drvFlags** to set to a one. So the program example

High Byte

Bit 8

Bit 0

00100100

00000000

The binary representation  
of the `drvFlags` parameter  
\$2400 (16-bits)

Bit Position 2 of the Byte  
Bit Position 5 of the Byte

## Device Manager Calls that the Desk Accessory can respond To

\$2400 in binary form represents a 00100100 00000000. This binary number, (using page 20 as a guide), shows that the driver can respond to control events and that the driver needs time to perform periodic action. [Hence bit settings in the `drvFlags` argument determine characteristics of the driver. -Ed.] The next argument `drvDelay` contains a tick count indicating how often the control routine should be called. A tick count of 0 means that the control routine will be called as often as possible. A 1 means that the action should be called every 60th of a second, 2 means every 30th of a second. Using a value of 60 in the program, our control routine will be called every full second. Remember whether this action occurs is dependent on how long it takes the application program to call `SystemTasks()`. `DrvEvmask` is a mask that filters what type of events the desk accessory will respond to, and is explained on page 16 of the Event Manager. Basically, just add up all the numbers that represent those events you want to respond to. The argument `drvMenu` represents the resource ID number of a menu that your desk accessory uses. You have seen many desk accessories that have a menu appear on the menubar such as Extras. To attach resources such as menus and windows, a special formula is used in determining the resource ID number. This formula is discussed on page 10 of the Resource Manager. In our example program, a resource number of -15424 is used. From the chart on page 10, we know that bits 15 and 14 are set to a 1. This means that all resources of desk accessories will have negative value, such as the above example. Next, bits 13, 12, and 11 are set to 0. This specifies that a **DRV** resource owns the attached resources. Bits 5 thru 10 contain the resource id number of the **DRV**. As mentioned in the very beginning of the program listing, we are using a resource id = 30. Bits 4 thru 0 are variable and allow us to have several menus attached to the same desk accessory. So the value -15424 = 11000011 11000000 in binary form. We used a variable of 0 in this **MENU** resource, however if you want to add another menu you can use a 1, then 2, and so on. **Length** and **Window title** are arguments that are used for the title of the window and the length of the window title.

## Format of the Desk Accessory File

The internal format of the desk accessory file can be found on page 12 of the Desk Manager. This is the final form that all desk accessories will have when finally compiled into machine code. If you have a compiler that creates desk accessories, then this information is just for interest only.

## Opening the **DRV** routine

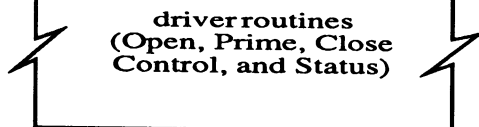
When a desk accessory is used for the first time, information about the desk accessory is read into a structure in memory called a **Device Control Entry** block. An illustrated version of this can be found on page 21 of the Device Manager. Looking at it, we notice that it has information about the window, menu, number of ticks, etc. This is very important, for this is how data can be accessed from one routine to the other. The first thing that is declared is a `*dctl` and a `*pb` pointer after the `accopen(dctl,pb)` function. The pointers allows access to the Device Control Block, and since this is a RAM driver (i.e. loaded from disk into memory), a paramunion block must be used to gain access to events and other information. Next, a menu structure and a window pointer are created, since this desk accessory will have a window and a menu associated with it. Then you check to make sure that that this desk accessory has not been already executed by checking for the presence of the window. Once the IF statement is true, use the function `getmenu` to get the menu resource that is already in the system file. Since the device control entry block already contains the menu resource id, just pass the value to the function `getmenu`. Next create the window, set the port, and assign the text mode. Now we get the device reference number (`dctlrefnum`) and insert it into the `windowkind` field of my window structure. This reference number is what the Device Manager uses internally instead of using the name of the Desk Accessory. Now assign the `deltwindow` field in the device entry block, a pointer that points to my window. At this point you have created all the menus and windows and the **Open** routine is finished.

## Controlling the **DRV** Routine

The Control routine represents the heart of the desk accessory. This routine is executed every time a **SystemTask** is called from the application program. This desk accessory creates a window, menu, and prints the time of day in the window.

Following the program, create `*dctl` and `*pb` pointers to the various information structures of the driver (i.e. desk

|         |                    |
|---------|--------------------|
| byte 0  | drvFlags (word)    |
| byte 2  | drvDelay (word)    |
| byte 4  | drvEMask (word)    |
| byte 6  | drvMenu (word)     |
| byte 8  | drvOpen (word)     |
| byte 10 | drvPrime (word)    |
| byte 12 | drvCtl (word)      |
| byte 14 | drvStatus (word)   |
| byte 16 | drvClose (word)    |
| byte 18 | drvName (byte)     |
| byte 19 | drvName + 1 (byte) |



#### Driver Flags

Number of Clock Ticks between periodic actions

Desk Accessory Event Mask

Menu ID of menu associated with desk accessory

Offset to the Open routine of the desk accessory

Offset to the Prime routine of the desk accessory

Offset to the Control routine of the desk accessory

Offset to the Status routine of the desk accessory

Offset to the Close routine of the desk accessory

Length of the name of the Desk Accessory

The actual name of the Desk Accessory

### File Structure of a Desk Accessory

accessory.) Then we create a new menu structure and window pointer. Using the `dctl` (device control block) we can extract the menuhandle and the pointer to my desk accessory's window, which are somewhere floating out in memory. Next comes the event structure that determines what the desk accessory will do each time the procedure `SystemTask` is called from the application program.

The action taken by the Control routine is determined by a message stored in `.cscode` field of the `paramunion.cntrlparam` structure. This field is accessed by the `pb` pointer that was created when the Control routine was just executing. The messages that we can expect to receive can be found on page 14 of the Desk Manager. These messages simply tell the Control routine what kind of action to take. In our example we used the `accrun`, `accevent`, and the `accmenu` messages. As outlined in the page 14, `accrun` handles periodic actions, such as getting the time for a clock. For handling events such as keyboard and mouse events, the `accevent` message is used. When using menus the Desk Manager uses the `accmenu` message.

Looking at the `accrun` portion of the switch statement, in the program listing, we can see that the time of day is inserted into the `datetime` variable, then printed onto the window that is associated with the desk accessory. This is done as long as the switch condition finds the `accrun` message in the `.cscode` field.

The `accevent` portion of the switch statement is a little more complicated since we must determine what type of event has occurred. Besides the `accevent` message, the control routine receives in the `csparam` field of the `paramunion` structure, a pointer to an event record. This event record is needed to tell us the what and where of the event. To get the

what of the event record, we do a 'C' type cast of the pointer, casting the pointer as an event record. One note, that in the header file of the `paramunion` structure that we must use a `long` that is a field of the `csparam` structure. This will allow us to conveniently get the address when type casting the pointer. (It may seem somewhat complicated but it works quite well.) (Back to the program.) Now that we know what type of event occurred, we can find out where it occurred, such as in a mousedown event. Using the same technique as above, we can pass to the `findwindow` routine where the mousedown event occurred.

Remember that all windows created by desk accessories are considered system windows, as talked about on page 4 of the Window Manager. The `insyswindow` constant is what `findwindow` will return, if the mousedown event occurred inside of our desk accessory window, so beware. If the person clicked inside of an application window, the desk manager will handle it and does not have to be addressed by the desk accessory. In our program when clicking inside of the system window the procedure `closedeskacc` is called. The only argument that is passed is the reference number of the desk accessory. This can be accessed thru `dctlrefnum` field of the device control block. The procedure `closedeskacc` terminates the Control routine of the desk accessory and starts the execution of the Close routine.

When selecting a desk accessory's menu the Desk manager generates an `accmenu` message. Once again we can extract the data using the `csparam` field. This allows us to get the item number of which item was chosen in the desk accessories menu. If we selected the first item we can hide or show the window on the desktop. Notice this also toggles the menu from show to hide and back again. If we select close then the `closedeskacc` procedure is called and the desk accessory is terminated.

## Closing the DRVr Routine

When the Closing the desk accessory the windows, menus and any other data structures must be disposed of. Then the `dctlwindow` pointer field of the device control block must have a `nil` inserted. Looking at our Close routine example we once again access the `pb` and `dctl` thru the use of pointers. This allows us to dispose of the window structures and menu resources. We redraw and update the menubar, then do a `sysbeep` just to let us know that the routine is complete and the desk accessory is no more.

## Some Closing Comments

Once you have compiled the desk accessory and the resource file, using REdit is useful for cutting and pasting the `DRVr` and `MENU` into the System file. Remember that the IDs of the `DRVr` and `MENU` are equal to 30. If you wish to change the `DRVr` id number, then remember to change the resource ids of the attached resources.

After looking over the code for this desk accessory you may decide that this is a trivial program. The idea here is to create a shell version that is understandable and most importantly works. In later issues we can approach the cutting and pasting between applications and desk accessories, and how to create a calculator.

Remember that desk accessories can be as powerful as application programs. It is not unlikely that a terminal program or text editor couldn't be written as a desk accessory. I write desk accessories because they can give the Macintosh a state of 'pseudo-concurrency', and because the Mac is an interesting machine to program.

```
*
*
* shell.r
* resource file
* resource id = 30
*
shell.rsrc
```

```
Type MENU
,-15424
Shell
 Hide Window
 (-
 Close
```

```
/*
shell.c
Wesley C. Swannack
shell for most desk accessories
written in Megamax C.
id number of the 'DRVr' = 30
*/
#include <global.h>
#include <acc.h>
#include <desk.h>
#include <qd.h>
#include <event.h>
#include <res.h>
#include <misc.h>
#include <mem.h>
#include <menu.h>
#include <te.h>
#include <font.h>
#include <file.h>
#include <win.h>
#include <control.h>
#include <device.h>

/* drvFlags, drvDelay,drvEvmask, drvMenu , Length and
window title */
ACC(0x2400,60, 0x014A,-15424,5, "Shell")
int menuflag = FALSE;

accopen(dctl, pb)
dctlentry *dctl; /* pointer to device
control block */
paramblockrec *pb; /* handle to parameter
block */
{
 menuhandle mymenu; /* creating my menu structure */
 windowpeek mywindow; /* pointer to the
window record */
 rect wr; /* starting size for the
window */
 if (dctl->dctlwindow == NULL) /* No window
created*/
 {
 mymenu = getmenu(dctl->dctlmenu);
 insertmenu(mymenu, 0); /* appending the menu
to the menubar */
 drawmenubar();
 setrect(&wr, 100, 50, 400, 250);
 /* creating the window */
 mywindow = newwindow(NULL, &wr, "Window", 0,
rdocproc, -1L, -1, 0L);
 setport(mywindow);
 textmode(srccopy);

 mywindow->windowkind = dctl->dctlrefnum;
 /* getting neg number for my sys window */
 dctl->dctlwindow = mywindow; /* device window now
my window ptr */
 }
 return 0;
}
```

```

accclose(dctl, pb)
dctlentry *dctl;
paramblockrec *pb;
{
 menuhandle mymenu; /* declaring menu */
 windowptr mywindow; /* declaring window */
 mymenu = getmhandle(dctl->dctlmenu); /* getting handle to desk menu */
 mywindow = dctl->dctlwindow; /* getting desk window */
 dctl->dctlwindow = NULL;
 disposewindow(mywindow);
 deletemenu(dctl->dctlmenu);
 releaseresource(mymenu); /* disposing of the menu rsrc*/
 drawmenubar();
 sysbeep(2); /* just to let me know when closed */
 return 0;
}

```

```

accctl(dctl, pb)
dctlentry *dctl;
paramblockrec *pb;
{
 long datetime;
 char timestr[100];

 menuhandle mymenu; /* declaring menu */
 windowptr mywindow; /* declaring window */

 mywindow = dctl->dctlwindow; /* getting my window stuff */
 mymenu = getmhandle(dctl->dctlmenu); /* getting my menu stuff */

 switch (pb->paramunion.cntrlparam.cscode)
 {
 case accrun:
 setport(dctl->dctlwindow);
 getdatetime(&datetime); /* getting the time for this moment */
 iutimestring(datetime, -1, timestr);
 moveto((100-stringwidth(timestr))>>1,15);
 /* moving the pen to draw with */
 drawstring(timestr);
 break;
 case accevent:
 /* getting the event */
 switch (((eventrecord *)pb->paramunion.cntrlparam.csparam.eventaddress)->what)
 {
 case mousedown:
 {
 int wherepressed;
 /* Only because it is a really long statement to print, notice that the next to lines are really one line */
 wherepressed=findwindow(&(((eventrecord *)pb->paramunion.cntrlparam.csparam.eventaddress)-> where),
 &mywindow);
 switch(wherepressed)
 {
 case insyswindow: /* in my desk accessory window */
 closedeskacc(dctl->dctlrefnum);
 break;
 }
 }
 case keydown:
 sysbeep(3);
 break;
 }
 case accmenu:
 }
}

```

```

switch (pb->paramunion.cntrlparam.csparam.menustatus.itemnumber)
{
 case 1: /* selected 'Hide' or 'Show' */
 if (menuflag)
 {
 selectwindow(mywindow); /* window is now visible */
 showwindow(mywindow);
 setitem(mymenu,1,"Hide Window");
/* inserting new menu item */
 /* user will see the 'Hide' option in the menu */
 menuflag = FALSE;
 }
 else
 {
 hidewindow(mywindow);
 setitem(mymenu, 1, "Show Window");
/* inserting new menu item */
 /* user will see the 'Display' option in the menu */
 menuflag = TRUE;
 }
 break;
 case 3:
 closedeskacc(dctl->dctlrefnum);
 break;
}
}
return 0;
}

```

```
accprime()
```

```
{
}
```

```
accstatus()
```

```
{
}
```

/\* I have included the appropriate data structures so that you can tell whether or not your files have the appropriate variables. The variables talked about are in boldface type

This is for reference only

```
*/
```

```

typedef union { /* control information */
 int sndval;
 int asncconfig;
 struct { } asyncinbuff;
 struct { } asyncshk;
 struct { } printer;
 struct { } fontmgr;
 ptr diskbuff;
 long eventaddress; /* address for desk events */
 long asyncnbytes;
 struct { } asyncstatus;
 struct { } diskstat;
 struct { /* menu manager */
 int menuident;
 int itemnumber;
 } menustatus;
 } opparamtype;
}

```

```
typedef opparamtype *opparamptr;
```

```

typedef struct {
 ptr iolink;
 int iotype;
 int iotrap;
 ptr iocmdaddr;
 procptr iocompletion;
 int ioresult;
 char *ionameptr;
 int iovrefnum;
 union {
 struct { } ioparam;
 struct { } fileparam;
 struct { } volumeparam;
 struct {
 int filler3;
 int cscode;
 opparamtype csparam;
 } cntrlparam;
 } paramunion;
 } paramblockrec;
}

```





# C Workshop

## PICT Rotation with Copybits



Robert B. Denny  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 12

A few years ago, I was studying the Smalltalk-80 language when I came across a novel image rotation algorithm [see Goldberg & Robson, *Smalltalk-80: The Language and its Implementation* (Addison Wesley, 1983), pp408-410]. It typifies the sorts of things that result from applying mathematics to software design.

The C Workshop is due for some less advanced articles, so here's the first. It serves two purposes: to illustrate a simple application which uses a runtime library, and to demonstrate QuickDraw bit-maps and the powerful CopyBits() toolbox service. We'll do this with an implementation of the image rotation algorithm.

### A Simple Application

Most C language systems provide glass teletype simulation, where one or more windows can act like TTY's via the "standard library" (stdio) calls such as scanf() and printf(). This makes it possible, in many cases, to build and run C applications which use text commands and output on the Mac.

If the stdio library permits access to the window record for the TTY window, you can let the library open the window, then experiment with QuickDraw services on that window. You don't have to start out with a "full-blown" application which uses event loops, textEdit, etc. The thing to remember is that *the run-time library turns C into a high-level language which has easy access to the toolbox*. In this column we use the Consulair Mac-C system; most other systems have similar support in their libraries.

Here is the classic "Hello World" program in Mac-C, with added QuickDraw commands to draw nested boxes in the glass teletype window:

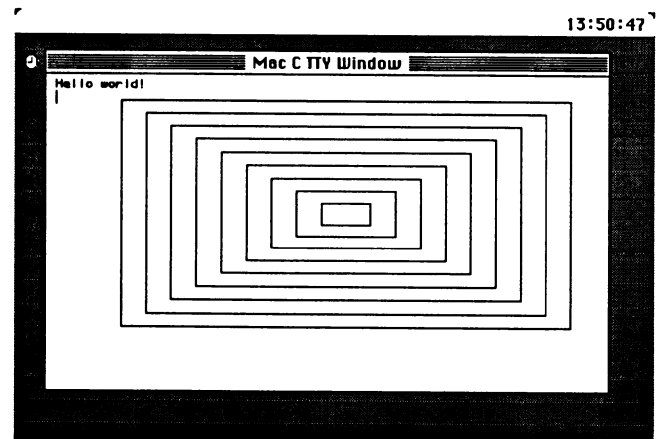
```
#include "Lib:stdio.h" /* Normally required */
#include "Lib:MacDefs.h" /* Toolbox */
#include "Lib:QuickDraw.h"
#include "Lib:Window.h"

main()
{
 Rect frame;
 int j, k;
 char buff[8];

 printf("Hello world!\n");
 for(j = 10; j < 100; j += 10)
 {
 k = 2 * j;
 SetRect (&frame, 240 - k, 110 - j, 240+k, 110 + j);
```

```
 FrameRect (&frame);
 }
 gets(buff); /* Wait for <Ret> then exit */
} /* End of program */
```

In this application, the Mac-C library opens a glass teletype window for use as stdout. It's the only window, so it's always the current "port". Thus, QuickDraw operations will be performed on the contents of that window. You don't need direct access to the window data. Here's how the screen looks:



### QuickDraw Bit Maps

A window is a special case of a QuickDraw *port*. Most QuickDraw services operate on the "current" port as established by a call to SetPort(). In the above application, the run-time library creates the window, and makes its contents the current port. By the time main() is called by the library, the window contents are ready for drawing.

A port is described by a data structure called a *grafPort*. One of the components of a grafPort is a small data structure called a *bitMap*. Contrary to common usage, a bit map is *not* the memory area which contains the image bits (pixels). Rather, it describes that area. The image is kept in a *bit image*. In C, the definition of a bitMap is:

```
struct __BM
{
 char *baseAddr; /* -> bit image */
 unsigned rowBytes; /* MUST BE EVEN */
 Rect bounds; /* Boundary Rectangle */
}

typedef struct __BM bitMap
```

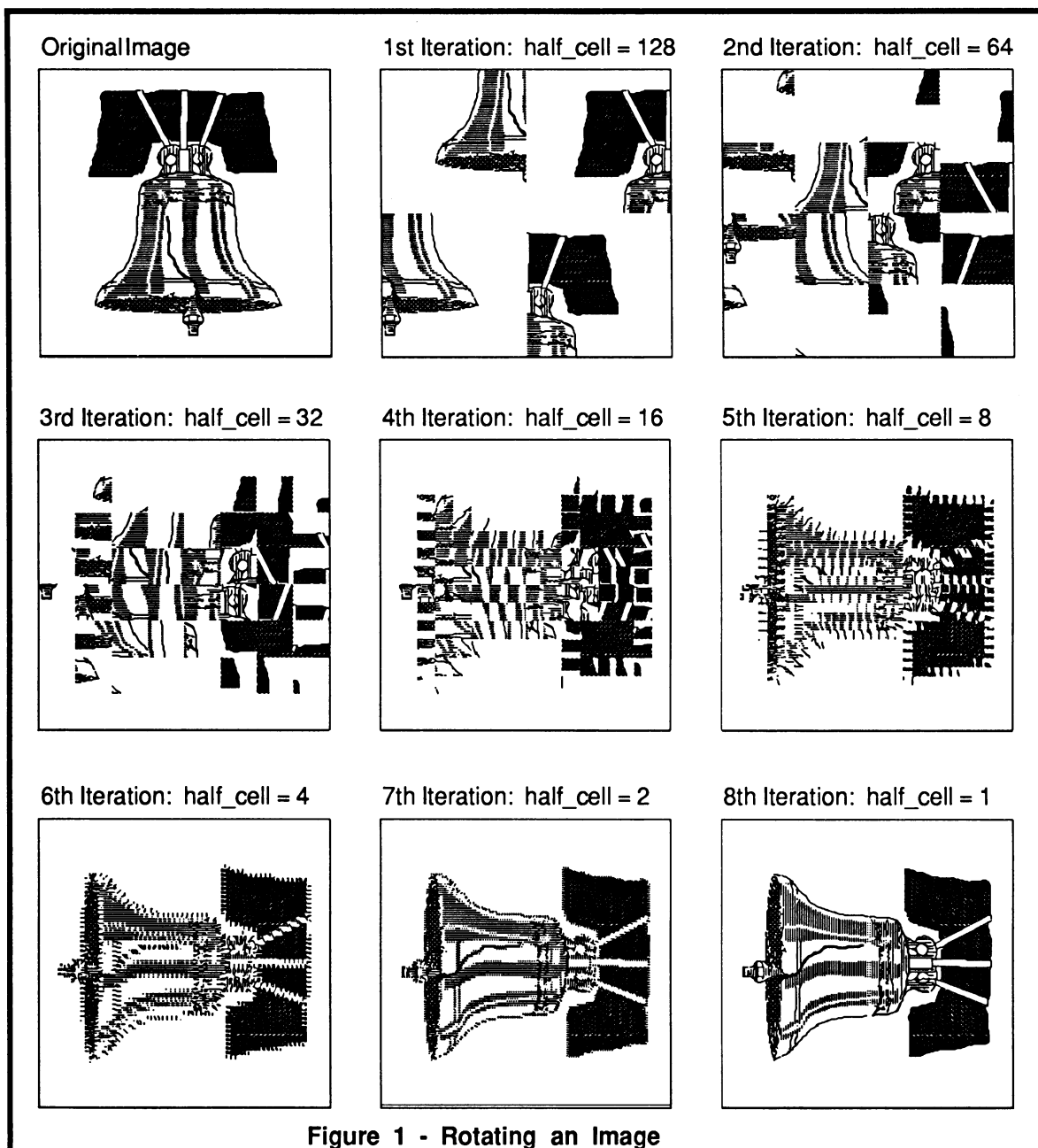


Figure 1 - Rotating an Image

The baseAddr field points to the memory address of the bit image. For ports whose bit images are on the screen, baseAddr always contains the address of the screen memory (\$7A700 on 512K Mac). It is possible to "draw" into a bit image other than the screen. Simply set up a grafPort with a bitmap pointing to your own bit image area, make it the current port, and draw away. This is how the printing system works; the image is drawn into a small bit image a strip at a time, and then transmitted to the imagewriter. The same commands draw to any bit image.

The rowBytes field describes the geometry of the bit image, in particular, the number of bytes that make up each row of the bit image. The value must be even; each row must be made up of an integral number of 16-bit words. This seems to

imply that bit images must always be a multiple of 16 bits wide. This is not the case, as we'll see now.

The bounds field establishes the actual dimensions of the bit image and the "local" coordinate system for the image. There is no restriction that the upper left corner of a bit image must be at coordinates (0,0) or that the origin coordinates must be positive. The dimensions of the bounds rectangle set the "logical" size of the bit image, as opposed to the "physical" size indicated by rowBytes and the total size of the bit image array.

It's possible to work with on-screen windows without coming into direct contact with bitMaps. The image rotation algorithm uses the "blitting" service CopyBits() to perform manipulations involving bitMaps. The image being rotated is

kept in a screen bit image so you can observe the rotation while it's in progress. Just remember that the window record contains the grafPort which contains the bitMap, which describes the "chalkboard" bit image.

### Image Rotation Using CopyBits

Ever wonder how MacPaint can rotate images so fast? Rotation of an image by a multiple of 90 degrees is a useful operation, and it's easier than you might think. The algorithm about to be described uses a clever sequence of bit transfer operations to rotate a square image whose sides are  $2^n$  bits in length, for integral  $n$ . It can be generalized to operations on arbitrary rectangles, as is done in MacPaint. Note how MacPaint does not rotate about the "center" of an image.

The algorithm consists of a series of permutations on the image. The left and right halves are swapped, followed by an exchange of the upper-left and lower-right quadrants of the swapped image. Then the four quadrants themselves are permuted in the same way, then divided into quadrants and the resulting sixteen cells are permuted, etc. The process completes when the cell size has reduced to two by two bits.

If you wrote a routine to implement the above method literally, it would take geometrically increasing time as the bit image size increased. Specifically, for an image  $2^n$  bits on a side, it would take

$$n_{\text{permutations}} = 1 + 4 + 16 + \dots + 4^{n-1}$$

to complete the rotation. This would render the algorithm a laboratory curiosity.

The amazing feature of the actual algorithm is its ability to perform all cell permutations for a given cell size *in parallel*! This means that execution time is proportional to  $\log_2(n)$ , where the image is  $2^n$  bits on a side. The penalty is that storage is required for two additional bit images the same size as the image to be rotated.

Figure 1 shows what it looks like when the Liberty Bell (Copyright 1984, T/Maker Graphics, used with permission) is rotated as a 256 x 256 bit image. Each pixel is mapped to a 2-pixel square on the LaserWriter used to produce MacTutor. The rotation takes  $\log_2(256) = 8$  steps.

Now let's get down to business. Besides the image itself, the algorithm requires two bit images the same size as the transform image. One, called *mask*, contains 1's in the upper left quadrant of each cell, 0's elsewhere. The other, called *temp* is used for scratch storage.

Each major step in a permutation is accomplished via a series of bit-transfer (blit) operations involving the three bit images. On the Mac, blitting is done via the powerful CopyBits() service, which can copy a bit image with coordinate translation and size scaling. The major steps in a permutation are (1) swap left and right cell halves, (2) exchange lower-right and upper-left quadrants of each cell and (3) refine the mask in preparation for the next cell subdivision.

Figures 2, 3 and 4 show the bit transfers needed for the major steps of the first permutation. Each bit transfer is a

single CopyBits() operation. The gray areas show the destination rectangles for each CopyBits(). Keep in mind that the destination is clipped to the bounds in its bitMap.

### Adventures With CopyBits()

The image rotation demo program described below makes heavy use of the QuickDraw CopyBits() service. Success followed several adventures which I'll describe. Well, I won't describe the first adventure ... I'm embarrassed. The other two adventures relate to the first blit of the mask refinement phase, where the entire mask is copied and shrunk into the upper-left 128 by 128 bit quadrant (see Fig. 4).

First, I was relieved to discover that CopyBits() works when the source and destination are the same bit image, as long as the destination Rect is to the left and above the source Rect. Specifically, mask refinement step 1 takes a single CopyBits(). That's the good news.

Now for the bad news. I discovered another feature (read that as "bug") in CopyBits(). Mask refinement starts by copying the entire mask into the upper left quadrant. The image is reduced to one half its size. Most of the time.

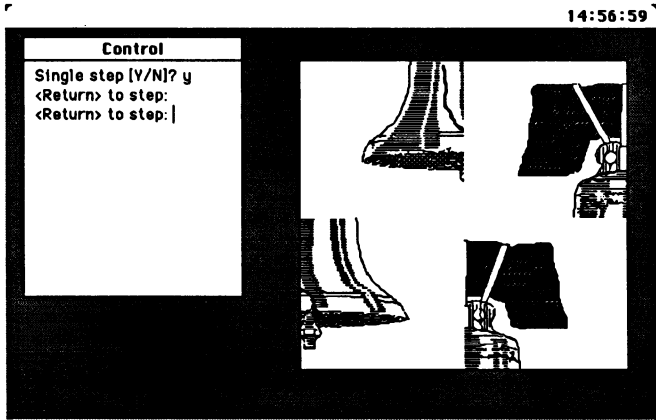
The first permutation uses a mask with a single 128-bit black square in the upper left quadrant, which should shrink to a 64-bit black square on the first refinement blit (see Fig. 4 again). It doesn't. Instead, the resulting black rectangle is 65 bits on a side. *The scaling feature of CopyBits does not scale accurately or consistently*, even if the scaling factor is a power of 2! This "small" error caused the rotation algorithm to fail miserably.

Fortunately, the solution is simple. Fake CopyBits by supplying a source Rect that is one bit larger on the right and bottom edges, making the reduction slightly more than 50%. Ugly, yes, but it works & anything more general or rigorous is likely to be a lot more complex. It's not a panacea, so beware. *Don't take image reductions made by CopyBits() for granted; test the results for accuracy.*

### A Demonstration Program

We finish up with the C code for a demonstration of the algorithm. The program was used to produce the Liberty Bell transformation images of Figure 1. It may be used to transform any PICT resource that will fit within the 256 by 256 bit area. Simply paste the image into the scrapbook and then cut it out using the Resource Editor. Save it in a resource file, then change the program code to open that resource file and get the proper resource by its ID. For the sample, I put the T/Maker Liberty Bell into a resource file named "Bell" with a resource ID of 1024.

The program opens up 2 windows, one for TTY simulation, the other for displaying the image during rotation. The *altStart* entry point is used to suppress the default TTY window. Later the setTTY() function hooks the "custom" TTY window up to stdout. The screen looks like Figure 5 (next page) after the first permutation.



Before the Liberty Bell is loaded into the image window, the window is used to draw the initial mask image. The mask and temp images don't need a grafPort, so they are described by simple bitMaps. This means that QuickDraw can't be used to draw into them directly. So the "seed" mask is drawn into the image window with FillRect(), then blitted to the mask image, after which the window is erased in preparation for loading the PICT.

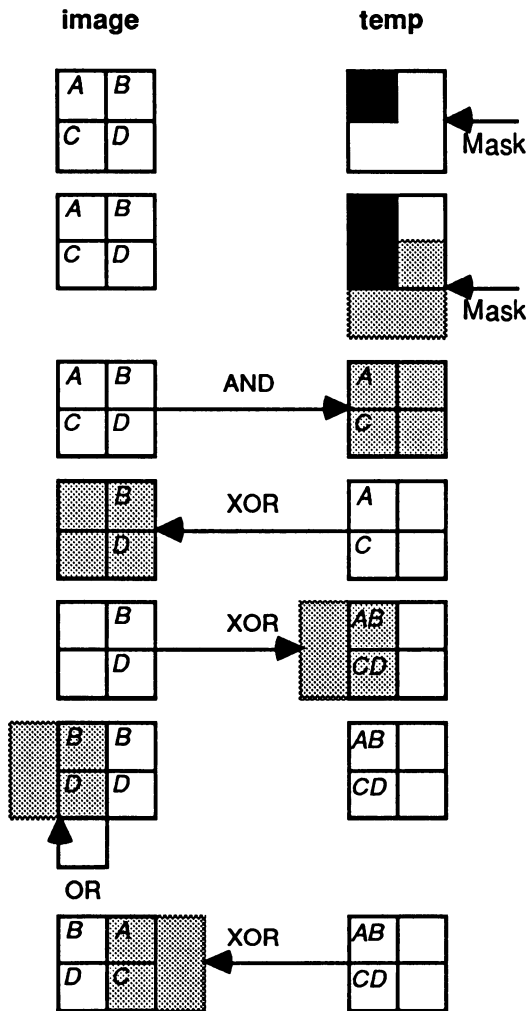


Figure 2 - Swap halves

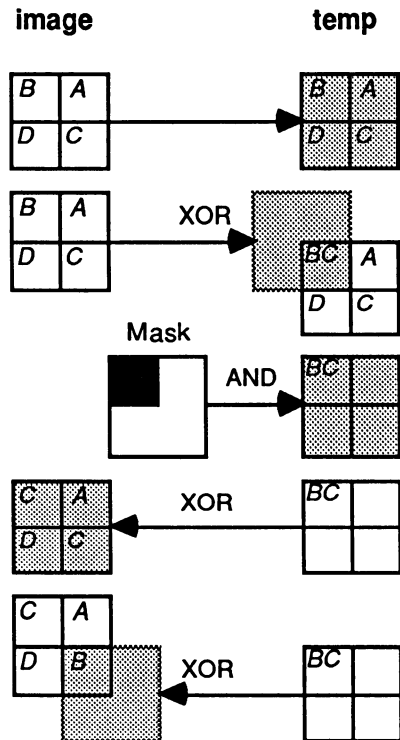


Figure 3 - Exchange Quadrants

Next, the resource file is opened and the PICT-1024 resource is read in with GetResource() and locked down. Then DrawPict() is called to paint the picture into the image window.

Before the picture is drawn, the PICT's bounds Rect is used to calculate the border needed to center it in the 256 by 256 bit image. Afterward, the PICT is unlocked and disposed. Finally, the rotation is performed, optionally in steps controlled by gets() calls.

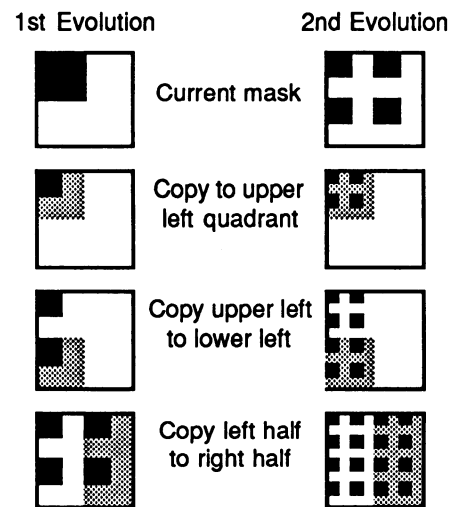


Figure 4 - Mask Evolution

```

/*
 * Rotate.C - Demonstrate Smalltalk Image Rotation Algorithm
 *
 * Written by:
 * Robert B. Denny, Alisa Systems, Inc.
 * September, 1985
 *
 * LINKER COMMAND PROCEDURE (Consulair Mac C V4.0, either Consulair or MDS linker)
 * --
 * /Output Dev:Rotate
 * /Type 'APPL' '????'
 * Dev:Rotate.REL
 * Lib:Standard Library
 *
 * $
 * --
 *
 * Copyright (C) 1985, MacTutor Magazine
 *
 * Permission granted to use only for non-commercial purposes.
 * This notice must be included in any copies made hereof.
 * All rights otherwise reserved.
 *
 * Warning: This code was edited for publication which could have introduced minor errors
 */

/*
 * Included files
 */
#include "Lib:Stdio.h" /* Required when using stdio library */
#include "Lib:MacCDefs.h" /* Has Consulair specials */
#include "Lib:MacDefs.h" /* Basic toolbox definitions */
#include "Lib:QuickDraw.h" /* QuickDraw structs and consts */
#include "Lib:Window.h" /* Window Manager structs & consts */

/*
 * Definitions & Parameters
 */
#define plainDBox 2 /* My H files must be old ... */
#define noGrowDocProc 4 /* ... these names match Inside Mac */
#define srcAnd notSrcBic /* A more reasonable name */
#define TRUE 1 /* I use the following everywhere */
#define FALSE 0
#define byte unsigned char
#define word unsigned int
#define longword unsigned long

/*
 * The following definitions control key aspects of the program, which you may change. You only need
 * change these definitions.
 */
#define PICT_FILE "Bell" /* Name of resource file containing PICT */
#define PICT_ID 1024 /* Resource ID of PICT */

/*
 * Static Variables
 */
static BitMap maskBits = {0, 32, 0, 0, 256, 256}; /* Mask bitmap - baseAddr set at runtime */
static BitMap tempBits = {0, 32, 0, 0, 256, 256}; /* Temp bitmap - baseAddr set at runtime */
static Rect hackRect = {0, 0, 257, 257}; /* Used in CopyBits hack (see article text) */

static WindowPtr imageWindow; /* --> image window's grafPort & data */

```

```

static Rect iwRect = {45, 231, 301, 487}; /* Location of image window on screen */

static WindowPtr ttyWindow; /* --> TTY window's grafPort & data */
static Rect twRect = {45, 15, 235, 185}; /* Location of control window on screen */

/*
 * Main Program
 */
main()
{
 /*
 * Automatics
 */
 char buf[80]; /* Used to receive gets() responses */
 word step; /* TRUE means single step transformation */
 Rect xRect, yRect; /* Scratch Rect's used in rotation */
 PicHandle srcPict; /* Handle to PICT resource */
 word picWidth, picHeight; /* PICT dimensions, pixels */
 word half_cell; /* Half-cell size, pixels */
 BitMap *ibp; /* Often used pointer to image window bitMap */
 Rect *irp; /* Often used pointer to image window portRect */

 /*
 * Begin code
 */
 HideCursor(); /* No mouse used here */

 /*
 * Create custom TTY window, clear it out & attach to stdout
 */
 ttyWindow = newWindow(0, &twRect, "007Control", TRUE, /* Make TTY window */
 noGrowDocProc, -1, FALSE, 0);
 SetTTY(ttyWindow); /* Hook this window up to stdout */
 ClearTTY(ttyWindow); /* Clear it out, just for fun */

 /*
 * Create the image window and make pointers to its bitMap and portRect, used often below.
 */
 imageWindow = newWindow(0, &iwRect, 0, TRUE, /* Make image window */
 plainDBox, -1, FALSE, 0);
 ibp = &(imageWindow->portBits); /* --> image window's bitMap */
 irp = &(imageWindow->portRect); /* --> image window's portRect */

 /*
 * Allocate bit image space for temp and mask, fill in bitMaps with the array addresses.
 */
 maskBits.baseAddr = (Ptr)calloc(4096, sizeof(word)); /* Allocate mask bit image area */
 tempBits.baseAddr = (Ptr)calloc(4096, sizeof(word)); /* Allocate temp bit image area */

 /*
 * Paint starting mask into image window, then transfer to mask bit image. Erase
 * image window when done.
 */
 PenNormal(); /* Reset graphics pen to black, etc. */
 SetPort(imageWindow); /* Prepare to draw in image window */
 SetRect(&xRect, 0, 0, 128, 128); /* Upper left quadrant */
 PaintRect(&xRect); /* Fill it in with black. Now have initial mask */
 CopyBits(ibp, &maskBits, irp, &(maskBits.bounds), srcCopy, 0); /* Transfer mask pattern to its bit image */
 EraseRect(irp); /* Clear out the image window */

 /*
 * Load the PICT, compute target Rect to center in image window, then draw it there.

```

```

*/
OpenResFile (PICT_FILE);
srcPict = GetResource ('PICT', PICT_ID);
HLock (srcPict);
irp = &((*srcPict)->picFrame);
picWidth = (irp->right - irp->left);
picHeight = (irp->bottom - irp->top);
xRect.left = (256 - picWidth) >> 1;
xRect.top = (256 - picHeight) >> 1;
xRect.right = xRect.left + picWidth;
xRect.bottom = xRect.top + picHeight;
DrawPicture (srcPict, &xRect);
HUnlock (srcPict); /* Unlock the resource, we no longer need it */
DisposHandle (srcPict);

/* Open the resource file */
/* Load PICT resource */
/* Lock resource ... */
/* Pointer to PICT's bounds Rect */
/* Compute PICT width */
/* Compute PICT height */
/* Form PICT-size Rect centered in image window */
/* (Could do this with InsetRect ()) */

/* Draw picture into target Rect in image window */

/*
 * Hook up "stdout" to our custom TTY window and query user about single stepping
 */
SetTTY(ttyWindow);
puts("Single step [Y/N]? ");
gets(buf);
puts("\n");
step = (toupper(buf[0]) == 'Y') ? TRUE : FALSE;

/* Attach stdout to TTY window */
/* Pop the question */
/* Get the answer */
/* Echo newline (Consulair quirk) */
/* Translate answer to boolean */

/*
 * Begin the rotation algorithm
 */
irp = &(imageWindow->portRect);
half_cell = 128;
while(half_cell)
{
 /* --> image window portRect -- used often */
 /* Initialize half-cell dimension */
 /* Main loop */

 /* If single-stepping */

 {
 /* Wait for <return> to step */
 puts("<Return> to step: ");
 gets(buf);
 puts("\n");
 }

 SetRect (&xRect, 0, 0, 256, 256);
 SetRect (&yRect, 0, 0, 256, 256);
 /* Initialize scratch rectangles to image coord's */

 /*
 * Phase 1 - Swap left and right cell halves
 */
 CopyBits (&maskBits, &tempBits, &xRect, &xRect, srcCopy, 0);
 OffsetRect (&yRect, 0, half_cell);
 CopyBits (&maskBits, &tempBits, &xRect, &yRect, srcOr, 0);
 CopyBits (ibp, &tempBits, irp, &xRect, srcAnd, 0);
 CopyBits (&tempBits, ibp, &xRect, irp, srcXor, 0);
 OffsetRect (&yRect, -half_cell, -half_cell);
 CopyBits (ibp, &tempBits, irp, &yRect, srcXor, 0);
 CopyBits (ibp, ibp, irp, &yRect, srcOr, 0);
 OffsetRect (&yRect, half_cell << 1, 0);
 CopyBits (&tempBits, ibp, &xRect, &yRect, srcXor, 0);

 /*
 * Phase 2 - Exchange lower-right and upper-left cell quadrants
 */
 wait(500);
 /* Delay to allow seeing each phase */
 CopyBits (ibp, &tempBits, irp, &xRect, srcCopy, 0);
 OffsetRect (&yRect, -(half_cell << 1), -half_cell);
 CopyBits (ibp, &tempBits, irp, &yRect, srcXor, 0);
 CopyBits (&maskBits, &tempBits, &xRect, &xRect, srcAnd, 0);
 CopyBits (&tempBits, ibp, &xRect, irp, srcXor, 0);

```

```

OffsetRect (&yRect, half_cell << 1, half_cell << 1);
CopyBits (&tempBits, ibp, &xRect, &yRect, srcXor, 0);
/*
 * Phase 3 - Refine mask for next smaller cell size
 */
half_cell >>= 1;
/* Reduce cell size by 1/2 */
SetRect (&xRect, 0, 0, 128, 128); /* Refine mask */
SetRect (&yRect, 0, 128, 128, 256);
CopyBits (&maskBits, &maskBits, &hackRect, &xRect, srcCopy, 0);
CopyBits (&maskBits, &maskBits, &xRect, &yRect, srcCopy, 0);
SetRect (&xRect, 0, 0, 128, 256);
SetRect (&yRect, 128, 0, 256, 256);
CopyBits (&maskBits, &maskBits, &xRect, &yRect, srcCopy, 0);
}
puts("<Return> to exit: ");
gets(buf);
} /* END OF PROGRAM */

```

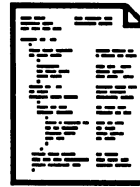




# C Workshop

## Try Pop-Up Menus!

C



Mike Schuster  
MacTutor Vol. 1 No. 13

### What's a Pop-Up Menu?

The Macintosh™ user interface provides a repertoire of graphic objects that perform functions when pressed with the mouse. Pressing a scroll bar's arrow moves the document under the window. Selecting a tool from a palette determines the type of object to be drawn. Pressing on a menu title presents a list of choosable actions and attributes. A feature missing from this repertoire is the ability to associate a menu with an arbitrary graphic object on the screen. A pop-up menu provides just this capability.

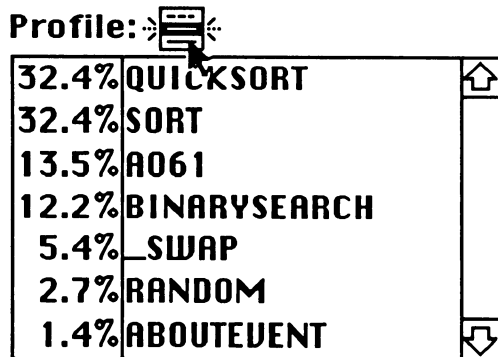


fig. 1 Arbitrary Icon Menu

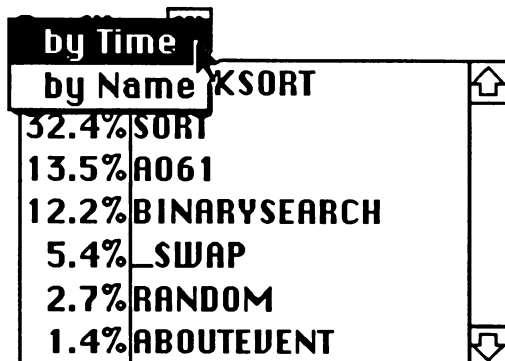


fig. 2 Icon pops up a menu!

Here's how it works: When you press on a object with an associated pop-up menu (the little menu icon above), the menu instantly "pops-up" right on top of the object and under the pointer. While holding down the mouse button, you move the pointer down the menu. As pointer moves to each item, the item is highlighted. The item that's highlighted when you release the mouse button is chosen. As soon as

the button is released, the command blinks briefly, the menu disappears, and the command is executed, just like a pull-down menu.

A pop-up menu shares many of the advantages of a pull-down menu: It's invisible until you want to see it, yet at the same time it's easy to see and choose items from. Until you choose an item, nothing happens, so you can look at a pop-up without making a commitment to do anything.

The advantage of a pop-up is that it directly associates actions and attributes with an object on the screen. Its biggest disadvantage is that there may be no indication that a given object has an associated menu, until you happen to press on it. This disadvantage could lead to less transparent and less consistent applications, especially if the application's standard menus were replaced with pop-ups. Don't do that! I find them most useful in dialog boxes and desk accessories.

One final issue. When the user presses on an object with a pop-up menu, it might be desirable to highlight the object and have the menu pop-up just below the object, in a manner similar to pull-down menus. I'd call such a scheme a "pop-down" menu, and leave its implementation as an exercise.

### Designing a Pop-Up

To make pop-up menus easy to use, I wanted to minimize the number of additional routines an application or desk accessory must call. Also, I wanted to apply all of the existing resource editing and compiling tools to construction and modification of the resource file description of a pop-up. Finally, to minimize my programming effort, I wanted to use as much of the Menu Manager's built in machinery as possible. With these goals in mind, I decided to use the standard menu record and the standard menu definition procedure without modification.

My efforts resulted in the single routine PopUpSelect. Its interface is almost identical to the Menu Manager routine MenuSelect:

```
typedef short int16;
typedef long int32;
int32 PopUpSelect(theMenu, hitPt)
MenuHandle theMenu;
Point * hitPt;
```

Here's how you use it: When your application receives a

mouse-down event in an object that has an associated pop-up, your application should call the `PopUpSelect`, supplying it with a handle to the pop-up menu and the point where the mouse button was pressed. `PopUpSelect` will pop-up the menu, track the mouse and highlight menu items until the user releases the mouse button. `PopUpSelect` returns a 32 bit integer in a manner identical to `MenuSelect`. The high-order 16 bits contains the menu ID of the pop-up menu, the low-order 16 bits contains the menu item number of the item that was chosen. If no item was chosen, the value returned is 0. (I use `int16` and `int32` to avoid the ambiguity inherent in the sizes of C's short, int, and long types. Also, I pass points by address rather than by value, in deference to Inside Mac.)

Before `PopUpSelect` can be called, the pop-up menu itself must be set up. Since a pop-up has the same structure as a pull-down menu, you do this by reading the menu from a resource file using `GetMenu`, or allocate it with `NewMenu` and filling it with items using `AppendMenu`, `AddResMenu` or `InsertResMenu`. The only difference in usage between pull-down and pop-up menus is that pull-downs are added to the menu bar using `InsertMenu`, and pop-ups are passed to `PopUpSelect` whenever the appropriate mouse-down events occur.

One thing to note: Don't bother defining a command key equivalent to an item in a pop-up. Since the pop-up is not in the menu bar, the Menu Manager routine `MenuKey` won't be able to find it. You could add all of your pop-ups to the menu bar just before the `MenuKey` call, and remove them just after, but don't do this! There's no need for these items to have command equivalents.

### Making It Work

`PopUpSelect` is built up from the following sequence of operations:

- determine where to draw the pop-up
- save the part of the screen under the pop-up
- draw the pop-up
- track the mouse until the mouse button is released
- blink the selected item, if any
- erase the pop-up and restore the screen
- return the appropriate result

The first task, that of determining where to draw the pop-up, is based on the size of the menu and the location where the mouse was pressed. The size of the menu is defined by the fields `menuWidth` and `menuHeight` in the menu record (whose definition is shown below), which specify the horizontal and vertical dimensions of the menu, in pixels. The Menu Manager determines these values when the menu is allocated and filled with items.

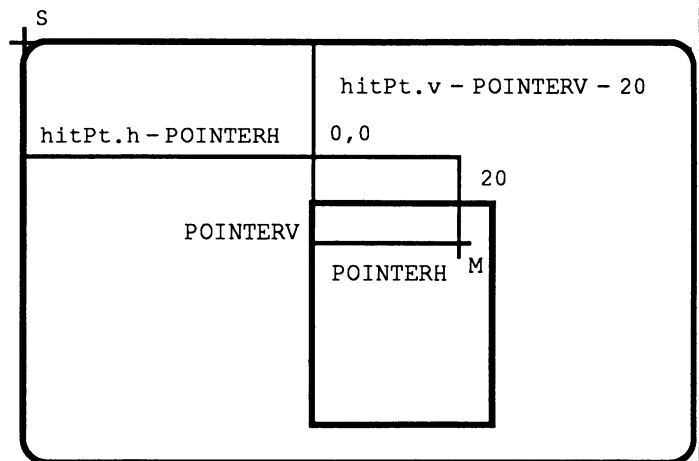
typedef struct

```
{
 int16 menuID;
 int16 menuWidth;
 int16 menuHeight;
 Handle menuProc;
 int32 enableFlags;
 Str255 menuData;
} MenuInfo, *MenuPtr, **MenuHandle;
```

The standard menu definition procedure draws the menu's items inside a rectangle of these dimensions. The black border and shadow of the standard pull-down menu appear immediately outside that rectangle.

I decided that the rectangle should be positioned so that the location where the mouse was pressed is just inside of the top-right corner of the menu. The pop-up hangs generally downward and to the left of the pointer. The only constraint on this positioning is that the menu must be completely visible in the desktop portion of the screen. Thus, the rectangle must be shifted in the appropriate direction if the mouse location is near the menu bar or an edge of the screen.

The technique I used to determine the coordinates of the menu rectangle can best be explained with the aid of a diagram. The point labeled S is the top-left corner of the screen. The point labeled M is the location where the mouse was pressed (defined by the `hitPt` argument). The rounded corner rectangle represents the screen. The thick bordered rectangle containing M represents the boundary of the pop-up.



Notice that top-left corner of the menu rectangle is displaced `POINTERH` pixels horizontally and `POINTERV` pixels vertically from M. These values incorporate my decision to have the menu appear generally to the left and downward from the pointer. They are constants in the implementation, and hence are easy to change.

The top of the menu rectangle is displaced 20 pixels

downward from the local origin, labeled 0,0 in the diagram. By defining a local origin in this manner, the standard menu definition procedure is tricked into thinking that it is drawing the menu just below the menu bar. That is, the top coordinate of the menu rectangle in the local system is 20, the height of the standard menu bar. Given these values, you should be able to find that the horizontal coordinate of S is `-(hitPt.h - POINTERH)` and its vertical coordinate is `-(hitPt.v - POINTERV - 20)`.

The final constraint on the local location of S is that the menu must be completely visible in the desktop portion of the screen. This constrains the horizontal coordinate of S to be less than or equal to 0 (otherwise the left side of the menu would be off-screen) and greater than or equal to the width of the menu minus the width of the screen (otherwise the right side would be off-screen). A similar situation holds vertically. All of these computations (plus an allowance for the menu's frame and shadow) are incorporated in the arguments of a Quickdraw `SetOrigin` call, which sets up the desired local origin.

One final item to note: `SetOrigin` translates the coordinates of the current port's `portBits.bounds`, `portRect`, and `visRgn` to the new coordinate system, but not its `clipRgn`. Hence, a call to `ClipRect` with the translated `portRect` as an argument is appropriate after the `SetOrigin` call.

`PopUpSelect` accomplishes the task of saving the part of the screen under the pop-up by copying that part to an off-screen bitmap using the Quickdraw routine `CopyBits`. The off-screen bitmap is allocated with a `NewHandle` call. The width of the menu rectangle (plus the width of the frame and shadow) determines the value of its `rowBytes` field (appropriate rounded up to a multiple of 2 bytes, since `rowBytes` must be even). The height of the menu rectangle (plus the height of the frame and shadow) determines the number of rows in the bitmap. These two values multiplied together, plus the size of a bitmap structure, determine the number of bytes to allocate. The bitmap structure itself is built in the first few bytes of the allocated area. The `baseAddr` field of the bitmap points to the byte immediately following the end of the bitmap structure.

The next task, that of drawing the menu, is performed by calling the standard menu definition procedure whose handle is contained in the `menuProc` field of the menu record. A menu definition procedure has the following interface:

```
#define mDrawMsg 0
#define mChooseMsg 1
#define mSizeMsg 2

MenuDefProc(message, theMenu, menuRect,
 hitPt, whichItem)
 int16 message;
 MenuHandle theMenu;
 Rect *menuRect;
 Point *hitPt;
```

```
int16 *whichItem;
```

Passing the message `mDrawMsg` to the menu definition procedure causes it to draw the menu inside the rectangle `menuRect`. The Window Manager port should be the current port when this message is sent. I set the port's `clipRgn` equal to `menuRect`, just to be safe.

The task of tracking the mouse is accomplished by passing the message `mChooseMsg` to the menu definition procedure. When the procedure receives `mChooseMsg`, its `hitPt` parameter is the current mouse location, and its `whichItem` parameter is the item number of the last item that was chosen from the menu. If the location is in an enabled menu item, the procedure unhighlights `whichItem` and highlights the newly chosen item (unless the new item is the same as the `whichItem`), and returns the item number of the new item in `whichItem`. If the location isn't in an enabled item (or isn't inside `menuRect`), the procedure unhighlights `whichItem` and returns 0. The following fragment of code shows how this works:

```
whichItem = 0;
SetPt(hitPt, 0, 0);
do
{
 MenuDefProc(mChooseMsg, theMenu,
 &menuRect, hitPt, &whichItem);
 GetMouse(hitPt);
}
while (WaitMouseUp());
```

`WhichItem` and `hitPt` are both initialized to 0 since the local coordinate system has changed, invalidating the original value of `hitPt`. This initialization is consistent since the local origin is outside `menuRect`. `GetMouse` returns the mouse location in the local coordinate system of the current port, which is just what we want. Finally, `WaitMouseUp` is called rather than `StillDown` so that the pending mouse-up event is removed, to avoid confusing the application (`MenuSelect` does this too).

If the final value of `whichItem` is not zero, then `PopUpSelect` interrogates the menu blink value in the system parameter ram area and sends the menu definition procedure the `mChooseMsg` message an appropriate number of times to blink the selected item. The `hitPt` argument is alternately modified so that the item blinks.

The task of restoring the screen after the mouse-up occurs is easily accomplished with `CopyBits`. The off-screen bitmap is deallocated after `CopyBits` completes. The original coordinate system is also restored with a second `SetOrigin` call.

Finally, `PopUpSelect` returns the appropriate result based on the menu's `menuID` field and the final value of `whichItem`.

## An Implementation

The remainder of the article contains the implementation of PopUpSelect and a sample program showing how to use it. The sample draws the caution alert icon on the screen and pops-up a short menu whenever the icon is pressed. The sources are set up to use the Consulair C compiler and header files, version 4.01. [Note that a complete Megamax version is supplied along with this Consulair version on the source code disk, courtesy of the author. -Ed.]

```
/*
 * Pop-Up Menu Selection
 * by Mike Schuster
 *
 * function PopUpSelect(theMenu: MenuHandle, hitPt: Point): Longint
 *
 * with hitPt equal to the point (in global coordinates) where the mouse
 * button was pressed, PopUpSelect will pop up theMenu and retain control
 * by tracking the mouse and highlighting menu items until the mouse button
 * is released. PopUpSelect returns a 32-bit integer in a manner identical
 * to the Menu Manager routine MenuSelect.
 *
 * October 16, 1985: Version 1.0
 */

#include "memory.h"
#include "resource.h"
#include "quickdraw.h"
#include "menu.h"
#include "osmisc.h"

#define GETSYSPPTR ((SysPPtr) 0x1F8)

typedef short int16;
typedef long int32;

#define mDrawMsg 0
#define mChooseMsg 1
#define mSizeMsg 2

#define POINTERH (*theMenu)->menuWidth - 4 /* width offset from hitPt */
#define POINTERV 8 /* height offset from hitPt */

#define MENUV 20 /* height of menu bar */

#define FRAMEH 2 /* width of menu frame */
#define FRAMEV 2 /* height of menu frame */

#define DELAY 2L /* blink delay */

/* pin integer i between lower and upper bounds l and u */
static pin(i, l, u)
 int16 i;
 int16 l;
 int16 u;
 {
 if (i < l)
 return l;
 else if (i > u)
 return u;
 }
```

```

else
 return i;
}

/* invoke the standard menu definition procedure */
static MenuDefProc(message, theMenu, menuRect, hitPt, whichItem)
int16 message;
MenuHandle theMenu;
Rect *menuRect;
Point *hitPt;
int16 *whichItem;
{
 #asm
 move.w message, -(A7) ; push first parameter
 move.l theMenu, -(A7)
 move.l menuRect, -(A7)
 move.l hitPt, A0
 move.l (A0), -(A7)
 move.l whichItem, -(A7) ; push last parameter
 move.l theMenu, A0 ; get menu handle
 move.l (A0), A0 ; get menu pointer
 move.l 6(A0), A0 ; get menu proc handle
 move.l (A0), A0 ; get menu proc pointer
 jsr (A0) ; dive in
 #endasm
}

/* popup menu selection routine */
int32 PopUpSelect (theMenu, hitPt)
MenuHandle theMenu;
Point *hitPt;
{
 GrafPtr port; /* current graf port */
 GrafPtr wMgrPort; /* window manager port */
 BitMap **theMenuBits; /* handle to BitMap to save screen in */
 BitMap *menuBits; /* pointer to "" */
 int16 rowBytes; /* rowBytes of "" */
 int16 rows; /* rows of "" */
 Rect menuRect; /* menu rectangle, in local coordinates */

 int16 whichItem; /* selected item */
 int16 blink; /* blink count */
 int32 nilPt; /* nil point for blink */

 /* return if mouse is not down */
 if (!WaitMouseUp())
 return 0L;

 /* determine the menu rectangle, in local coordinates */
 SetRect(&menuRect,
 0, MENUV, (*theMenu)->menuWidth, MENUV + (*theMenu)->menuHeight);

 /* inset the menu rectangle to include the frame and shadow */
 InsetRect(&menuRect, -FRAMEH, -FRAMEV);

 /* allocate the bitmap to save screen in */
 rowBytes = ((menuRect.right - menuRect.left + 15) >> 4) << 1;
 rows = menuRect.bottom - menuRect.top;
 theMenuBits = (BitMap **)
 NewHandle(rowBytes * rows + (int32) sizeof(BitMap));

```

```

/* return if no space */
if (!theMenuBits)
 return 0L;

/* lock down the bitmap */
HLock(theMenuBits);
menuBits = *theMenuBits;

/* initialize the BitMap */
menuBits->baseAddr = (char *) (menuBits + 1);
menuBits->rowBytes = rowBytes;
menuBits->bounds = menuRect;

/* save the current graf port, use the window manager port */
GetPort(&port);
GetWMgrPort(&wMgrPort);
SetPort(wMgrPort);

/* set origin so that menu definition procedure draws menu under hitPt */
/* pin so that whole menu is visible */
SetOrigin
 (pin(POINTERH - hitPt->h,
 (*theMenu)->menuWidth - wMgrPort->portRect.right +
 FRAMEH, 1 - FRAMEH),
 pin(MENUV + POINTERV - hitPt->v,
 MENUV + (*theMenu)->menuHeight - wMgrPort->portRect.bottom +
 FRAMEV, 1 - FRAMEV));

/* clip to save the screen */
ClipRect(&wMgrPort->portRect);

/* save the screen */
CopyBits(&wMgrPort->portBits, menuBits,
 &menuBits->bounds, &menuBits->bounds, 0, 0L);

/* erase and frame the menu rectangle */
InsetRect(&menuRect, FRAMEH, FRAMEV);
EraseRect(&menuRect);
InsetRect(&menuRect, -1, -1);
FrameRect(&menuRect);
InsetRect(&menuRect, 1, 1);

/* add shadow */
PenNormal();
MoveTo(menuRect.left + 1, menuRect.bottom + 1);
Line((*theMenu)->menuWidth, 0);
Line(0, - (*theMenu)->menuHeight);

/* clip for the standard menu definition procedure */
ClipRect(&menuRect);

/* prepare to call the standard menu definition procedure */
LoadResource((*theMenu)->menuProc);
HLock((*theMenu)->menuProc);

/* draw the menu */
whichItem = 0;
SetPt(hitPt, 0, 0);
MenuDefProc(mDrawMsg, theMenu, &menuRect, hitPt, &whichItem);

```

```

/* track the mouse until the button is released */
do
{
 MenuDefProc(mChooseMsg, theMenu, &menuRect, hitPt, &whichItem);
 GetMouse(hitPt);
}
while (WaitMouseUp());

/* blink the item */
if (whichItem)
{
 for (blink = GETSYSPPTR->Misc >> 2 & 0x3; blink; blink--)
 {
 nilPt = 0L;
 MenuDefProc(mChooseMsg, theMenu, &menuRect, &nilPt, &whichItem);
 nilPt = Delay(DELAY); /* Inside Mac and Consulair C differ here */
 menuDefProc(mChooseMsg, theMenu, &menuRect, hitPt, &whichItem);
 nilPt = Delay(DELAY); /* Inside Mac and Consulair C differ here */
 }
}

/* done calling the standard menu definition procedure */
HUnlock((*theMenu)->menuProc);

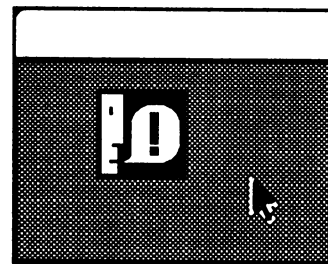
/* clip to restore screen */
ClipRect(&wMgrPort->portRect);

/* restore the screen and clean up */
CopyBits(menuBits, &wMgrPort->portBits,
 &menuBits->bounds, &menuBits->bounds, 0, 0L);
HUnlock(theMenuBits);
DisposHandle(theMenuBits);

/* restore the window manager port origin and the current graf port */
SetOrigin(0, 0);
ClipRect(&wMgrPort->portRect);
SetPort(port);

/* return the standard result */
return whichItem ? ((int32) (*theMenu)->menuID << 16) + whichItem : 0L;
}

```



Output from Sample Program

```

/*
 * Pop-up Menu Example
 */

```

```

#include "quickdraw.h"
#include "menu.h"
#include "events.h"

```

```

extern long PopUpSelect();

```

```

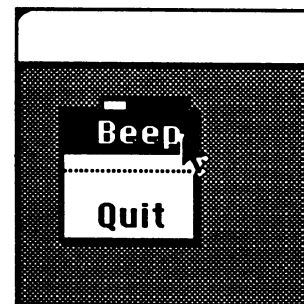
main()
{
 MenuHandle menu;
 EventRecord event;
 GrafPtr port;
 Rect box;

```

```

/* initialize the managers */
/* InitGraf(&thePort); */ /* Consulair does this for us */

```



Click makes menu pop up!

```

/* InitFonts(); */
/* InitWindows(); */
InitMenus();
TEInit();
InitDialogs(0L);
InitCursor();

/* draw the icon */
GetWMgrPort(&port);
SetPort(port);
ClipRect(&port->portRect);
SetRect(&box, 32, 32, 64, 64);
PlotIcon(&box, GetIcon(0));

/* initialize the popup menu */
menu = NewMenu(1, "");
AppendMenu(menu, "\pBeep;-;Quit"); /* \p for pascal string */

/* handle mouse down events */
while (1)
{
 GetNextEvent(everyEvent, &event);
 if (event.what == mouseDown)
 if (PtInRect(&event.where, &box))
 switch (LoWord(PopUpSelect(menu, &event.where)))
 {
 case 1:
 SysBeep(4);
 break;
 case 3:
 ExitToShell();
 break;
 }
 }
}

```





# Pascal Procedures

## MacPascal Arrives!



ah!s benchmark

Chris Derossi  
Editorial Board  
MacTutor Vol. 1 No. 1

Well, after waiting and waiting and waiting, Macintosh Pascal has finally arrived, and with its arrival, Mac users now have a serious high-level language for programming on the Macintosh. This column will help the users of Macintosh Pascal both to learn about Mac Pascal, and to learn how the wondrous abilities of the Mac ROM can tie into Mac Pascal for creating powerful and useful programs.

For this opening column, we'll be taking a general look at Pascal, and we'll see how you can get started right away learning how to use your Pascal to the fullest. Let us start out by examining the contents of your Pascal package.

The documentation is in three separate parts, with each section contained in its own little booklet. The first, and smallest book contains the standard instructions for using the normal Macintosh User Interfaces, to which Mac Pascal conforms. For experienced users of the Mac, this book will be of very little help; you already know all about clicking, dragging, and editing with the mouse. For the less experienced user, this book will teach you what you need to know in order to get started with Pascal right away.

The most important part of this book is the section which describes the use of the Observe and Instant windows. This is the only explanation of these windows found in the documentation. As we get further into Pascal, the importance of these windows will grow.

The booklet titled "Pascal Reference Manual" is an important tool for beginners and experienced programmers alike. Although it is not a tutorial, it does contain the entire syntax specification for Mac Pascal. Its primary use is as a look-up reference. New Pascal programmers may wish to go through the manual to get a general feel for the language, but as I'll discuss shortly, more can be learned by studying the example programs supplied with Mac Pascal.

The final book, "Macintosh Technical Appendix", is probably the most useful manual included. This manual describes the differences between Mac Pascal and both Lisa Pascal and ANS Pascal. In addition, this book has detailed information on two important areas of Mac Pascal: QuickDraw and SANE. Both of these will be covered in detail in further issues of MacTech.

Unfortunately, none of the documentation included with your Mac Pascal goes into any depth in the area of the Mac ROM, except for the QuickDraw section which is really more concerned with mathematical concepts. To learn about the language, and find out its limitations and its capabilities, we will be exploring a large variety of topics in this column. To begin with, however, a good source of information can be found in the example programs contained on your Pascal disk.

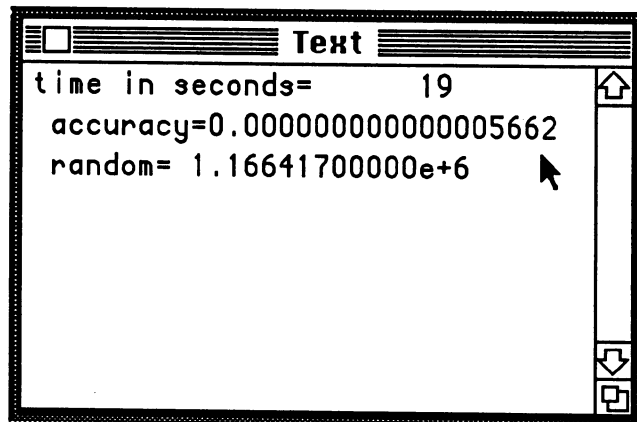


Fig. 1 Results

Figure 1 shows the output of the Ah!s benchmark published in creative computing. Notice the high degree of accuracy in only 19 seconds. This compares to MS Basic time of 1 minute, 36 seconds on the Mac. The random time does not compare due to the difference in implementation of the random number generator.

The Pascal disk contains a number of interesting programs that demonstrate some features of Mac Pascal. As you'll notice, most of the examples contain graphics, but some of them display the ability of Mac Pascal to do things like change window size, modify the cursor, interact with the mouse and the keyboard, and perform file I/O. Indeed, Mac Pascal has a wide range of abilities. Although many of the procedures and functions used in the examples are not clear, browsing through the included examples is a good way to get a feel for the power of Mac Pascal, and an idea of the range of functions available.

The other item that is included with your Mac Pascal disk is a second Mac Pascal disk. This is an exact copy for backup purposes. The backup disk is included because Mac Pascal is copy protected. The diskette cannot be copied, and the Pascal file cannot be moved to any other disks, including hard disk drives. The files, on the other hand, can be moved and should be copied.

Now that we've examined the things that are contained in your Mac Pascal package, let's say just a little bit about the interpreter itself. Before Mac Pascal, Pascal programs were compiled. That is, the Pascal program was written into a text

editor. Then, the Pascal compiler read the program text and converted it to a form that the computer could execute. The machine executable version was called object code, and the original text version was called source code.

If a program was to be modified, the text of the program had to be changed, and then the whole thing had to be compiled again. This could take lots of valuable time. On the other hand, the object code, since it was usually directly executable by the computer, ran very quickly. Macintosh Pascal, however, is not compiled; Mac Pascal is interpreted.

When we say that Mac Pascal is interpreted, we mean that the program is entered into a text editor which is part of the language interpreter. When the program is run, the interpreter reads the program and converts each line to an executable format. The individual lines are converted and run each time they are needed, and the converted lines are not kept available. This means that interpreted programs generally run slower than compiled programs. If the program is to be changed, though, it can be re-run immediately after modification. This decreases the amount of development time required.

In conclusion, Macintosh Pascal presents a useful and powerful programming environment for the Mac user. For the beginner, Pascal provides an excellent learning vessel, and for the experienced Mac user, Pascal provides a usable, structured interface to the internal magic of the Macintosh. This column will explore and enhance these qualities of Mac Pascal and will help you to better understand and take advantage of Mac Pascal.

#### **program Benchmark;**

{creative computing benchmark}

#### **type**

datetimerec = record

year, month, day, hour, minute, second, dayofweek : integer  
end;

#### **var**

a, r, s : extended;  
i, n : integer;  
result1, result2 : extended;  
begintime : datetimerec;  
endtime : datetimerec;  
t1, t2 : longint;

#### **begin {of main program}**

```

gettime(begintime);
for n := 1 to 100 do
begin
 a := n;
 for i := 1 to 10 do
 begin
 a := sqrt(a);
 r := r + random
 end;
 for i := 1 to 10 do
 begin
 a := a * a;
 r := r + random
 end;
 s := s + a
end;

result1 := abs(1010 - s / 5);
result2 := abs(1000 - r);
gettime(endtime);
t1 := begintime.hour * 3600 + begintime.minute * 60 +
 begintime.second;
t2 := endtime.hour * 3600 + endtime.minute * 60 +
 endtime.second;
writeln('time in seconds=', t2 - t1);
writeln(' accuracy=', result1 : 20 : 18);
writeln(' random=', result2 : 20 : 18);

end.
```



# *Pascal Procedures*

## *Introduction to QuickDraw*



*Chris Derossi*  
*Editorial Board*  
*MacTutor Vol. 1 No. 2*

One of the most valuable features of Mac Pascal is its ability to take advantage of the Macintosh's powerful ROM capabilities. To do this, Mac Pascal must be able to access the various routines that are contained within the Mac Toolbox. In this month's column, we will examine the interface between Mac Pascal and the ROM, and the necessary data structures for accessing quickdraw. Finally, a sample program is included which makes use of quickdraw's graphics routines.

In order to communicate to the Toolbox, several data structures have to be defined. These data structures are all presented in Pascal format because the routines contained in the Mac ROM are constructed for a Pascal environment. Unlike other programming languages, this makes data compatibility pretty straightforward for Mac Pascal users. But before we get into the details, let's briefly describe some of the theory behind QuickDraw.

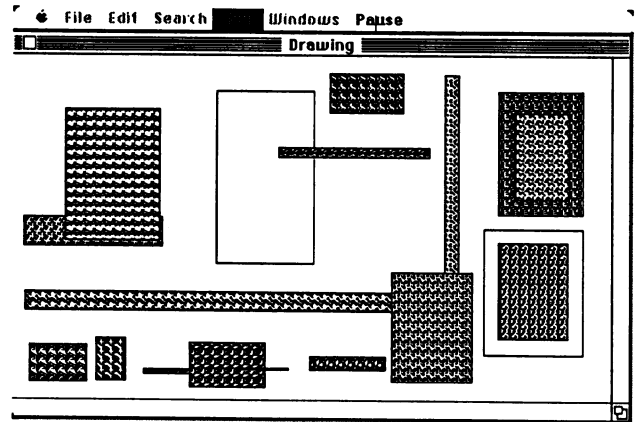
### **FOUNDATION OF QUICKDRAW**

QuickDraw graphics exist in cartesian coordinate planes. What this means is that graphics may be present on any portion of a continuous surface. The QuickDraw planes are finite, and extend from values -32768 to 32767 in both the vertical and horizontal directions. The smallest unit of distance that QuickDraw can work with is 1. An area of 1 by 1 is, therefore, a point. The boundary separating two points at sequential locations of the plane is of size zero. This means that two points placed next to each other are continuous, and have no space between them. Keep in mind that these are theoretical concepts, not screen bit-map specifications.

### **QUICKDRAW DATA STRUCTURES**

In Mac Pascal, the data structure 'Point' is a record with two components representing the coordinates of the upper left corner of a point. The two fields of Point are Point.h and Point.v which are the horizontal and vertical coordinates respectively. Each value is of type Integer.

The rectangle is a common object found in QuickDraw. Rectangles are frequently used to create windows, define drawing areas, and to provide parameters for manipulating other objects. A rectangle has four parameters. These are: Top, Left, Bottom, and Right. These parameters may be thought of as four integers, or as the two points TopLeft and BottomRight. In actuality, the second method is used, and the type 'Rect' is defined as a record with the two fields Rect.TopLeft and Rect.BotRight. Each of the fields is of type point.



Output from QD\_Demo

Another object defined with points is the line. Although there is no Mac Pascal data type for this object, it can be characterized by two points, for the beginning and ending positions of the line.

The final data type we will examine this month is the pattern. QuickDraw allows regions to be filled with patterns, as is done with the paint bucket in MacPaint. A pattern is a set of sixty-four points either on or off, arranged in an eight by eight square. To fill an area larger than eight by eight, the Macintosh repeats the pattern both vertically and horizontally as many times as necessary. The upper left corner of the pattern is aligned with the coordinate (0,0), and is independent of the coordinates of the area being filled.

Since a byte is eight bits, and a bit represents one point, patterns can be represented as a series of eight bytes. In Mac Pascal, the type 'Pattern' is an array [0..7] of 0..255, which is effectively eight sequential bytes. There are some predefined patterns including White, Black, and Grey.

### **DRAWING IN QUICKDRAW**

The drawing mechanism in QuickDraw is the Pen. The pen has modifiable properties which include location, width, height, and pattern. There are further properties which will be covered in a later column. A pen of width and height of one, and a pattern of black will always draw with one-point wide, black lines.

In addition to the many objects of QuickDraw, many actions are available to be performed on or with them. Most objects may be Framed, Painted, Filled, and Erased. The

QuickDraw actions are called QuickDraw verbs, and they are applied to objects. The verb Frame specifies that the outline of the given object should be drawn. The verb Paint means that the object is drawn completely filled. Each of these actions use the pen for drawing, and as such use the current pen pattern. The verb Erase does not use the pen pattern, but instead uses the pattern currently set for the background. Fill allows you to specify a pattern to use, and is therefore a little bit more versatile than Paint.

For example, FrameRect will draw the outline of a rectangle, while PaintRect will draw a solid rectangle. FillRect will draw a solid rectangle in the given pattern. EraseRect clears the given rectangle to the current background pattern.

Each verb-object combination is a procedure, and takes applicable parameters. The Rect procedures take a rectangle as their only parameter. The FillRect procedure needs a Rect first, and a Pattern second. The rectangle is drawn as a solid with the pattern.

A frequently encountered situation is one where the Macintosh needs to know if a given point is inside a particular rectangle. (i.e. is the mouse's position inside a certain window?) To satisfy this need, a function titled PtInRect returns a boolean of True if the point is inside the rectangle, and False if not. The parameters of this function are a point first, and then a rectangle.

### PROGRAMMING EXAMPLE

Now let us examine the example program that makes use of these data structures, procedures, and functions. The operation of the program is fairly simple and straightforward. The user is presented with a blank background and the mouse pointer. The user forms rectangles by pressing the mouse button when the pointer is at one of the rectangle's corners, and holds the button down while dragging the mouse to the opposite diagonal corner. In order to provide some feedback about the shape of the rectangle, a 'rubber banding' diagonal line follows the mouse until the button is released. When the button is released, the rubber band line is removed, and the rectangle is drawn.

### WHERE'S THE MOUSE?

After the rectangle is drawn, the user may place the mouse pointer inside the rectangle, and press the button. As long as the button is held, the Mac continually fills the rectangle with random, changing patterns. Once the mouse button is pressed outside of the rectangle, the pattern is frozen, and a new rectangle is begun. (Please see the sample output in fig. 1.)

The construction of the program is not complex. The variable declarations for the main program consist of three points and one rectangle. The points are used as work variables for reading the mouse location, and the rectangle is the one currently being drawn or filled.

The main program calls two procedures. The first one allows the user to create the rectangle, and provides the rubberbanding effect. The second one creates random patterns

and fills the rectangle.

Since the value of a rectangle's Top must be less than or equal to that of its bottom, two procedures called 'Min' and 'Max' help keep the points in the right order. The same holds true for the Left and Right values.

The key procedure calls in the Make\_Rect procedure are Button, GetMouse, DrawLine, PenPat, FrameRect, and InsetRect. The function Button returns True if the mouse button is pressed, and False if not. GetMouse uses two VAR parameters for obtaining the current horizontal and vertical position of the mouse pointer.

Since we need to draw and erase the rubberbanding line, we use the two pen patterns of Black and White. This has the disadvantage of erasing whatever the line passes over, but you can't have everything in a simple little program. The procedure PenPat sets the working pen pattern to the passed value.

DrawLine uses two points, this time in the form of four coordinates, to designate the starting and ending points of the line. DrawLine, of course, uses the current pen pattern.

Finally, after the mouse button is released, the rectangle is drawn with FrameRect. This is done with the pen pattern set to black. Since we don't want to draw over the outline with subsequent FillRect calls, the procedure InsetRect is called to change the coordinates of the rectangle by one on each side, making it slightly smaller.

After the rectangle is drawn, the Fill\_Rect procedure is called to fill it in. Note, please, that Fill\_Rect is distinct from FillRect. Fill\_Rect uses the calls Button, GetMouse, PtInRect, Random, and FillRect to perform its task. Button and GetMouse are used as before. PtInRect determines if the mouse is inside the new rectangle when the button is depressed.

A new pattern is created by calling the function Random eight times within a loop. Random returns a value from -32768 and 32767. Abs and Mod are used to bring the random number in the range 0..255. After the random pattern is created, FillRect is called to draw the pattern into the rectangle.

### MODIFYING THE PROGRAM

The program runs forever, and the menu option Halt must be chosen to exit. In order to overcome this, you might want to modify the program to present you with a Black or Grey box (read rectangle) where you'd click the mouse to quit. Where in the program would you draw the rectangle? At what point or points would you have to check to see if it has been clicked? How will you handle rectangles that overlap your 'exit' box? These exercises are left to the student!

That's it for this month. In the next column, we'll further explore the powerful features of QuickDraw, and find out what makes it so unique and wonderful among graphic tools. We'll also have another program that utilizes these terrific features. If you like our Pascal column or have programming ideas of your own to contribute, please feel free to write me care of MacTech. Ciao.

```
program MacTutor_QD_Demo;
```

```
{ This program will use the standard features of }
{ QuickDraw to introduce use of QuickDraw data }
{ structures and routine calls from MacPascal. }
{ -- By Chris Derossi }
```

```
var
 P1, P2, P3 : Point;
 WorkingRect : Rect;
```

```
function Min (Num1, Num2 : integer) : integer;
```

```
{ return the minimum of the two numbers passed. }
```

```
begin
 if Num1 < Num2 then
 Min := Num1
 else
 Min := Num2;
end;
```

```
function Max (Num1, Num2 : integer) : integer;
```

```
{ returns the maximum of two numbers passed. }
```

```
begin
 if Num1 > Num2 then
 Max := Num1
 else
 Max := Num2;
end;
```

```
procedure Make_Rect (var theRect : Rect);
```

```
{ This is the first of the two main procedure. It }
{ allows the user to select diagonal corners of the }
{ rectangle to be drawn, using a 'rubber band' line. }
{ When the button is released, the rectangle is }
{ drawn. }
```

```
begin
 repeat
 until Button; {Wait till the button is pressed}
 GetMouse(P1.h, P1.v); {Where was the mouse?}
 P2 := P1;
 P3 := P1;
 while Button do {update while button is down.}
 begin
 GetMouse(P3.h, P3.v);
 if (P2.h <> P3.h) or (P2.v <> P3.v) then
 {the mouse has moved}
 begin
 PenPat(White);
 DrawLine(P1.h, P1.v, P2.h, P2.v);
 {Erase the old line}
 PenPat(Black);
 DrawLine(P1.h, P1.v, P3.h, P3.v);
 {Draw the new line}
 P2 := P3;
 end;
 end;
 end;
```

```
 PenPat(White); {Erase line for the last time.}
 DrawLine(P1.h, P1.v, P2.h, P2.v);
 PenPat(Black);
 theRect.TopLeft.v := Min(P1.v, P2.v);
```

```
{Set the rectangle, making sure}
 theRect.TopLeft.h := Min(P1.h, P2.h);
{that the top is above the bottom}
 theRect.BotRight.v := Max(P1.v, P2.v);
{and the left is less than the}
 theRect.BotRight.h := Max(P1.h, P2.h); {right.}
```

```
 FrameRect(theRect); {Draw the rectangle}
 InsetRect(theRect, 1, 1);
{Shrink it so that the pattern does not draw}
{over the rectangle frame.}
end;
```

```
procedure Fill_Rect (theRect : Rect);
```

```
var
 myPat : Pattern;
 Index : Integer;
 Done : Boolean;
```

```
begin
 Done := False;
 while not Done do
 {continue until the mouse is outside of the}
 {new rectangle.}
 begin
 repeat
 until Button; {Wait for the button}
 GetMouse(P1.h, P1.v);
 if PtInRect(P1, theRect) then
 {is the mouse in the rectangle?}
 begin
 for Index := 0 to 7 do
 myPat[Index] := Abs(Random) mod 256; {make a
 random pattern}
 FillRect(theRect, myPat); {Fill the rectangle}
 end
 else
 Done := True;
 end; {While}
 end; {Fill_Rect}
```

```
begin {Main Program}
 while true do
 begin
 Make_Rect(WorkingRect);
 Fill_Rect(WorkingRect);
 end;
 end. {Main}
```



# Pascal Procedures

## Quickdraw does Regions!



Chris Derossi  
Editorial Board  
MacTutor Vol. 1 No. 3

In the last column, we were introduced to the simple access to QuickDraw from MacPascal. We found that the procedures for graphic drawing were easily called, using predefined routines and data structures. The concepts of cartesian coordinates, points, and lines were very important when dealing with QuickDraw. In this month's column, we will begin to explore the next step of QuickDraw complexity: the region.

Whereas a line (actually, a line segment) is a subset of points in one dimension, a region is a subset of points in two dimensions. When you draw a line with QuickDraw, the points from one endpoint of the line to the other are said to be part of the line. Points lying past either endpoint are not part of the line. Points off to the side of the line can never be part of the line, and are ignored; they are not part of the line.

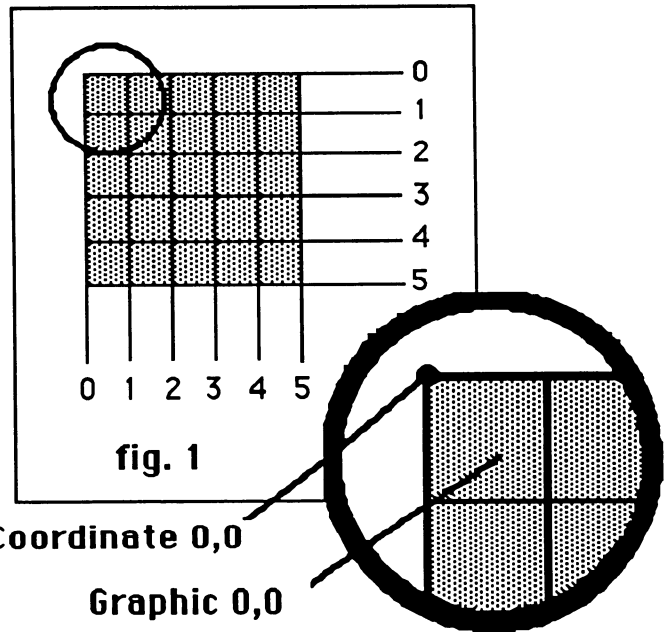
A similar concept is true for regions. When you have a region in a plane, the QuickDraw coordinate plane for example, some of the points are part of the region and the rest are not part of the region. There are no points that are both, and no points that are neither. Points that are part of the region are said to be inside the region, while the rest are outside the region.

The points inside a region do not have to be contiguous. That is, points or groups of points that comprise the region do not have to be adjacent. For the time being, we will call a group of points in the QuickDraw plane an area. A region may consist of zero, one, or several distinct areas.

For the most part, regions can be manipulated in the same manner as any other QuickDraw object. Just like circle and rectangles, regions can be framed, painted, erased, inset, offset, and filled. However, the marvelous aspect of regions is their shape. Regions can be any shape or combination of shapes, large, small, or empty. A region is empty when there are no points inside of it.

To describe or represent a region, its outline is defined. For example, to describe a circular region, one would only need to indicate the outline of the circle, not all of its interior points. Similarly, a rectangular region is represented as a framed rectangle. More complex shapes for regions can be defined using various combinations of simple shapes. A 'T' shaped regions could be described as the outline of two rectangles that just touch. It is important to realize, though, that the actual boundaries of a region are infinitely thin.

Because a region divides the plane into only two sets of points, those inside the region, and those outside the region, there can be no points that lie on the boundary of the region. For that reason, there is a distinction between coordinate points and graphic points. Coordinate points lie between graphic points and vice versa. Lines from one coordinate point



to another have no width, so the graphic points it separates are actually adjacent.

Illustration #1 shows some graphic points on a coordinate plane. Notice that a graphic point is to the 'right' and 'below' its corresponding mathematical coordinate. As mentioned before, any points with adjacent coordinates will be drawn on the screen as a single, solid area. The coordinate line separating them is only mathematical, and does not actually exist. A region's boundary is defined in terms of coordinate points, and no graphics points in the plane lie on the boundary.

When regions are defined in MacPascal, the procedure **OpenRgn** is first called to allocate the memory needed for the region. Then, the boundary of the region is defined by calling normal QuickDraw drawing routines, such as **FrameRect**, **FrameOval**, and **FrameRoundRect**. Only the outlines of the shapes are important; i.e. **PaintRect** acts just like **FrameRect**.

The outlines are summed together to form the boundary of the region. No drawing is performed because the boundaries are only mathematical, and do not involve real graphic points. When the entire outline is defined, a call to **CloseRgn** is made to stop the region definition. **CloseRgn** creates a handle to the new region, and passes it back in a variable of type **RgnHandle**. (Note: the variable must have first been initialized with a call to **NewRgn**.) All subsequent references

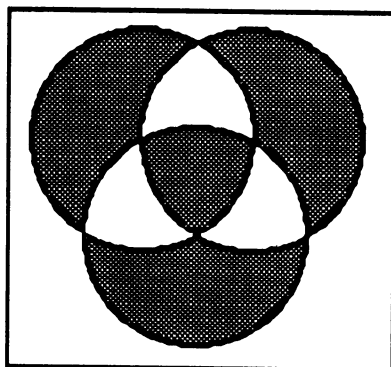


fig. 2A

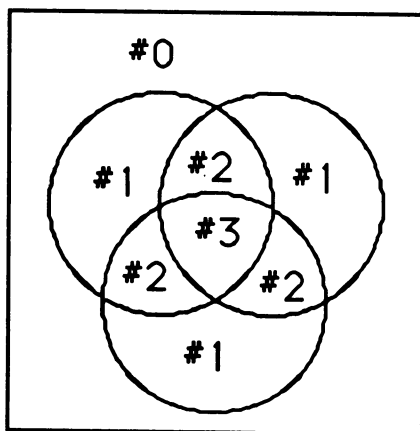


fig. 2B

to the region are done through its handle.

After `OpenRgn` has been called to start a new region, the region is empty. Then, areas are added to the region as the boundaries for those areas are 'drawn'. But, if a boundary encloses an area that is already part of the region, that area is changed to be outside of the region. If it is enclosed again, it is re-added to the region, and so on. In other words, areas are exclusive-ored with the existing region. Areas enclosed an odd number of times are part of the region, while areas enclosed an even number of times are not part of the region.

Illustration #2 shows a situation in which overlapping areas caused 'holes' in the region. Illustration #2a shows the region shaded, with the rest not shaded. Illustration #2b breaks the areas down. Area #0 was never enclosed, so it is not part of the region. Areas #1 were enclosed once each, and are part of the region. The areas numbered #2 were each enclosed twice, once from each of two circles; they are not part of the region. Finally, area #3 was enclosed three times and is part of the region.

It is frequently easier to define a region in terms of one area minus another instead of a sum of areas. For example, illustration #3 can be described as a series of four rectangles, or as one large rectangle minus the smaller, inside rectangle. The second way is easier and more intuitive. QuickDraw's method of exclusive-oring areas allows area to be added to and subtracted from regions.

Let us now put aside regions to talk about something else: clipping. **Clipping** is a graphics term that refers to drawing within certain boundaries. The easiest way to learn about clipping is to think of an example: anything drawn inside one window on the Macintosh should stay within that window and not get drawn outside of it. If the window is too small, the part of the drawing that is beyond the current edges of the window should not be drawn on the screen. The part of the drawing that is not actually drawn is said to have been clipped.

Clipping can be thought of as looking through the viewfinder of a camera. You can only see a portion of the world even though there is stuff beyond the edges of your sight. In order to emulate reality, and to create the effect of windows and such, everything drawn on the Mac is clipped, making each window and even the Mac screen a viewfinder on the whole picture.

There are two ways to do clipping. First, whoever is doing the drawing makes sure that nothing gets drawn beyond the appropriate boundaries. Second, anything desired is drawn without regards to boundaries, and clipping is controlled by the low level draw routines. This means that every point that



fig. 3a

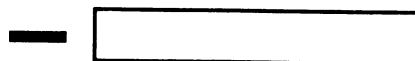
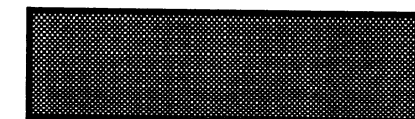


fig. 3b

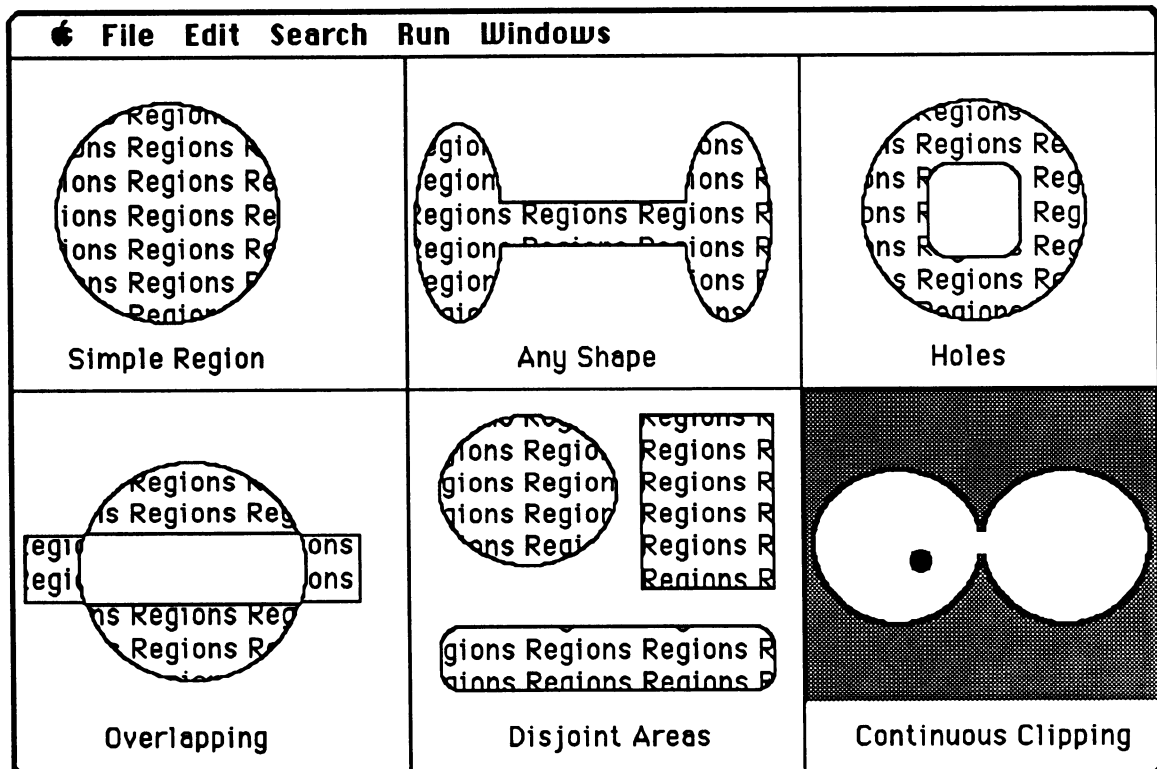


Fig. 4 Output

is drawn is checked for boundary limitations. Because this is done for every single point, it happens very often. If the clipping operation were slow, the graphic output on the Mac would take forever. The name QuickDraw is partially derived from its ability to handle operations like this very rapidly.

In general, this means that each separate window or drawing area on the Mac may consider itself the entire coordinate plane. QuickDraw makes sure that only the appropriate things are actually displayed, and that they fit within the proper bounds.

Many times, though, the clipping area is not rectangular like a window. (Clipping Area or Clipping Region refers to the areas where drawing occurs. The stuff outside of the clipping area is what is actually clipped.) Many times a window may be 'underneath' several other windows, and have only an unusually shaped portion visible. Nevertheless, the drawing in that window must be clipped to that unusual area. That's where regions come in.

Each window (or QuickDraw port) has an associated Clipping Region. It is this region that defines what will be displayed, and what will be clipped. For most normal windows, this region is simply rectangular, but when the window is partially obscured, the region may become more complicated. In addition, the clipping region of a window may be changed to suit the needs of the situation.

For example, you might want to fill or erase a region of a window without affecting anything else in the window. This could be done in pieces, or by setting the clipping region to the desired region and filling or erasing the whole window. Note that the clipping region only affects subsequent drawing, and has no effect on previously drawn graphics.

This month's MacPascal program is a demonstration of the use of regions, and the effects of using clipping regions. It shows clearly how to define regions, how to set the clipping region, and what happens when drawing is clipped.

Before we take a brief step-by-step look at the program, there are some general comments that should be made. First, in order to remain as compatible as possible with other Pascal/QuickDraw interfaces, calls that normally require a **Rect** parameter are not supplied four integers. Instead, **SetRect** is used to assign values to the rectangle's coordinates.

Second, when you type this program into your Mac and try to execute it, you might run out of memory; the data structures for regions take up lots of room. If this happens, try deleting the comments, closing all the windows, and ejecting unneeded diskettes to free up some memory. Users of 512K Macs should not have any problems.

The program begins by doing a 'uses QuickDraw2'. The library QuickDraw1 contains all the routines and data structures that we have used so far and is 'used' automatically. The routines and data structures for regions, however, reside in QuickDraw2 and must be included explicitly.

The **RgnHandle** that we use throughout the program is called **My\_Rgn**. Also, a **Rect** called **Big\_Rect** is declared. The constants **WordsPerLine**, **NumLines**, and **LineHite** are declared and set. These values will be used later to determine that amount and density of text to draw.

The main program calls only two procedures: **SetUp** and **MakeRgns**. **SetUp** puts away all of the MacPascal windows with **HideAll**, puts away the cursor with **HideCursor**, and displays the drawing window after setting its coordinates with



**SetDrawingRect.** In addition, **SetUp** draws three lines, dividing the window into six areas. **SetUp** also initializes **My\_Rgn** with **NewRgn** and assigns the coordinates of the drawing window to **Big\_Rect**.

**MakeRgns** is simply a series of calls to the six separate region demonstration procedures. The program was structured this way so that each region could be its own procedure and so modifications were easier. When all six region procedures have finished, the memory space used by **My\_Rgn** is deallocated with **DisposeRgn**, and the clipping region of the drawing window set back to full size.

The six region procedures each call upon three utility procedures called **DrawWords**, **Use\_Region**, and **ClearRgn**. **DrawWords** uses **DrawString** to draw the word 'Regions' several times according to the constants declared at the beginning. The final display of the words is clipped according to each different clipping region. **ClearRgn** sets the window's clipping region back to the full window so that drawing of titles, etc won't be clipped. **Use\_Region** draws the outline of the current **My\_Rgn** with **FrameRgn** and then shrinks it so that the outline lies outside of the region. This is to prevent later drawing from drawing over the outline. Then, the clipping region of the window is set to equal **My\_Rgn** with **SetClip**.

Five of the six region procedures follow the same general outline. First, **ClearRgn** is called to reset the clipping region. Next, a title is drawn for the region. Finally, the region is defined and used to clip the text created with **DrawWords**. Several aspects of regions and clipping are demonstrated.

The final region procedure is slightly different. As before, it resets the clipping region and draws a title. Then, it shades in the sixth rectangle with gray to provide for contrast later. Next, a region is defined and used in the same manner as the other five procedures. This region is then cleared to white, yielding a binocular-type window effect. After this is done, a small circle is animated. The circle is bounced in an imaginary rectangle that is somewhat larger than the region. The animation routine just continues the circle on its present course until it strikes a side of the imaginary rectangle. Then, a new course is chosen at random for it, and it continues.

Because the circle is sometimes not within the region, it is automatically clipped. This gives the effect of multiple planes that are stacked, with a window in the top one looking through to the bouncing ball. The clipping is performed automatically, and it is transparent to the animation routine. The animation ceases when the mouse button is pressed.

The power of **QuickDraw** can be seen when one considers the versatility of regions. The entire concept of multiple, independent windows, used extensively in the Macintosh, is based on the foundations of arbitrary clipping regions. Further, the **QuickDraw** clipping regions allow programs and applications to output to the screen regardless of the screen situation. This is a key factor in multi-tasking, and simultaneous application execution. (i.e. desk accessories.)

There are additional aspects of regions and **QuickDraw** that have yet to be covered, like calculations, ports, etc. The next column will focus on these final features, concluding our overall introduction to **QuickDraw**. We will find that this

basic background in **QuickDraw** will continue to aid us throughout the excursions into other areas of MacPascal such as windows, dialog boxes, and menus. Ciao.

**program Regions;**

```
{ Regions is a MacPascal demonstration program}
{ that provides six examples of the workings of}
{ the regions of QuickDraw. }
{ -- by Chris Derossi}
```

**uses**

```
QuickDraw2; {QuickDraw2 contains the stuff for regions.}
```

**const**

```
{ These constants determine the amount and density of the}
{ words used to show region clipping}
```

```
WordsPerLine = 6;
NumLines = 10;
LineHite = 15;
```

**var**

```
My_Rgn : RgnHandle; { This is our working region, used
 everywhere}
Big_Rect : Rect; { This is a rectangle that is the whole
 drawing window}
```

**procedure SetUp;**

```
{ SetUp clears the screen to a full sized drawing window and
 uses}
{ DrawLine to mark it off into six sections. It also inits our
 region}
{ and sets Big_Rect to the full window.}
```

**begin**

```
HideCursor;
HideAll;
SetRect(Big_Rect, 0, 20, 527, 357); {Full screen size}
SetDrawingRect(Big_Rect);
ShowDrawing; {Show only the drawing window}
DrawLine(0, 149, 527, 149);
DrawLine(176, 0, 176, 337);
DrawLine(352, 0, 352, 337);
My_Rgn := NewRgn; {Init our working region}
end;
```

**procedure DrawWords (X, Y : integer);**

```
{ DrawWords draw several lines of words starting at (X,Y)
 according}
{ to globally declared constants}
```

**var**

```
s : string;
a, b : integer;
```

**begin**

```
s := 'Regions ';
for a := 0 to NumLines do
```

```

begin
 MoveTo(X, Y + a * LineHite); {Start a new line in column Y}
 for b := 1 to WordsPerLine do
 DrawString(s);
 end;
end;

procedure Use_Region;

{ This procedure draws an outline of our region, then shrinks it
 so that}
{the outline is not in the region, and sets the drawing window's
 clipping}
{region to equal our region}

begin
 FrameRgn(My_Rgn); { Draw the outline of our region}
 InsetRgn(My_Rgn, 1, 1); { Shrink it}
 SetClip(My_Rgn); { Set the drawing window's clipping
 region}
end;

procedure ClearRgn;

{ In order to use the entire drawing window, this procedure sets
 the}
{clipping region to the whole window, using Big_Rect}

begin
 RectRgn(My_Rgn, Big_Rect); {Make our region a rectangle,
 screen size.}
 SetClip(My_Rgn);
end;

procedure DoRegion1;

{ This creates the first region, which is just a circle}

var
 r : Rect;

begin
 ClearRgn;
 MoveTo(25, 140);
 DrawString('Simple Region');

 begin {Create a region}
 OpenRgn;
 SetRect(r, 20, 20, 120, 120);
 FrameOval(r);
 CloseRgn(My_Rgn);
 end;

 Use_Region;
 DrawWords(0, 0);
end;

procedure DoRegion2;

{ This is region number 2. This is a concatenation of 3 shapes.}

var

```

```

 r : Rect;

begin
 ClearRgn;
 MoveTo(225, 140);
 DrawString('Any Shape');

 OpenRgn; {Start a new region}
 SetRect(r, 220, 65, 300, 85);
 FrameRect(r);
 SetRect(r, 180, 30, 220, 120);
 FrameOval(r);
 SetRect(r, 300, 30, 340, 120);
 FrameOval(r);
 CloseRgn(My_Rgn);

 Use_Region;
 DrawWords(176, 0);
end;

procedure DoRegion3;

{ Region number 3. Two overlapping regions form a 'hole'.}

var
 r : Rect;

begin
 ClearRgn;
 MoveTo(410, 140);
 DrawString('Holes');

 OpenRgn;
 SetRect(r, 380, 20, 480, 120);
 FrameOval(r);
 SetRect(r, 410, 50, 450, 90); {Completely inside the first
 shape}
 FrameRoundRect(r, 18, 18);
 CloseRgn(My_Rgn);

 Use_Region;
 DrawWords(352, 0);
end;

procedure DoRegion4;

{ Region 4. This creates a region which shows the effect of }
{partially overlapping region areas.}

var
 r : Rect;

begin
 ClearRgn;
 MoveTo(40, 310);
 DrawString('Overlapping');

 OpenRgn;
 SetRect(r, 30, 180, 130, 280);
 FrameOval(r); { Draw a simple circle }
 SetRect(r, 5, 215, 155, 245);
 FrameRect(r); { Draw a rectangle over the circle }
 CloseRgn(My_Rgn);

```

```

Use_Region;
DrawWords(0, 149);
end;

```

```

procedure DoRegion5;

```

```

{ Region 5. Regions do not have to be contiguous. This
 procedure creates}
{a region with three separate areas.}

```

```

var
 r : Rect;
begin
 ClearRgn;
 MoveTo(220, 310);
 DrawString('Disjoint Areas');

```

```

 OpenRgn;
 SetRect(r, 190, 160, 270, 230);
 FrameOval(r);
 SetRect(r, 280, 160, 340, 240);
 FrameRect(r);
 SetRect(r, 190, 255, 340, 285);
 FrameRoundRect(r, 18, 18);

```

```

 CloseRgn(My_Rgn);
 Use_Region;
 DrawWords(176, 150);
end;

```

```

procedure DoRegion6;

```

```

{ Region 6. This procedure illustrates the effect of continuous
 clipping}
{associated with any drawing, even animation.}

```

```

var
 r : Rect;

```

```

procedure Animate;

```

```

{ Animate, called only by DoRegion6, bounces a ball in the
 general area}
{of region 6, showing that the clipping occurs constantly, with
 now need}
{for special instructions. The animation stops when the mouse
 button is}
{pressed.}

```

```

var
 X, Y, dX, dY : integer;

```

```

begin
 X := 353; { Arbitrary starting position. }
 Y := 220;
 dX := 3; { Arbitrary beginning velocity. }
 dY := 0;
 repeat
 PenPat(white); { Use white for drawing to }
 PaintCircle(X, Y, 5); { erase ball at current position }
 X := X + dX; { upgrade position }

```

```

 Y := Y + dY;
 if (X < 353) or (X > 510) or (Y < 175) or (Y > 265) then
 begin { The ball has hit a "wall" and should bounce }
 X := X - dX; { Move the ball back to its last legal position }
 Y := Y - dY;
 repeat
 dX := ((random mod 3) - 1) * 7; { Choose new
 random velocities,}
 dY := ((random mod 3) - 1) * 7; {with each being -7,0, or 7 }
 until (dX < 0) or (dY < 0); { zero velocity is illegal }
 end
 else { new ball position is okay. }
 begin
 PenPat(black);
 PaintCircle(X, Y, 5); { Draw the ball in the new position }
 end;
 until Button; { Stop when the button is pressed }
 end;

```

```

begin { Region6 }

```

```

 ClearRgn;
 SetRect(r, 353, 150, 530, 290);
 FillRect(r, gray); { Use a gray background for contrast }
 MoveTo(375, 310);
 DrawString('Continuous Clipping');

```

```

 OpenRgn;
 SetRect(r, 357, 185, 432, 255);
 FrameOval(r);
 SetRect(r, 432, 185, 507, 255);
 FrameOval(r);
 CloseRgn(My_Rgn);

```

```

 Use_Region;
 SetRect(r, 353, 150, 530, 290);
 FillRect(r, white); { Erase the inside of the region, using auto-
 clipping}
 Animate; { Do the bouncing ball }
end;

```

```

procedure MakeRgns;

```

```

{ This is a brute force way to call all six regions, but allows us
 to break the regions down into separate procedures.}

```

```

begin
 DoRegion1;
 DoRegion2;
 DoRegion3;
 DoRegion4;
 DoRegion5;
 DoRegion6;
 ClearRgn;
 DisposeRgn(My_Rgn); { Free the memory used for region }
 ShowCursor; { We need the cursor! }
end;

```

```

begin { Regions }
 SetUp;
 MakeRgns;
end.

```



# *Pascal Procedures*

## *Ports O'Call in Quickdraw*



*Chris Derossi*  
*Editorial Board*  
*MacTutor Vol. 1 No. 4*

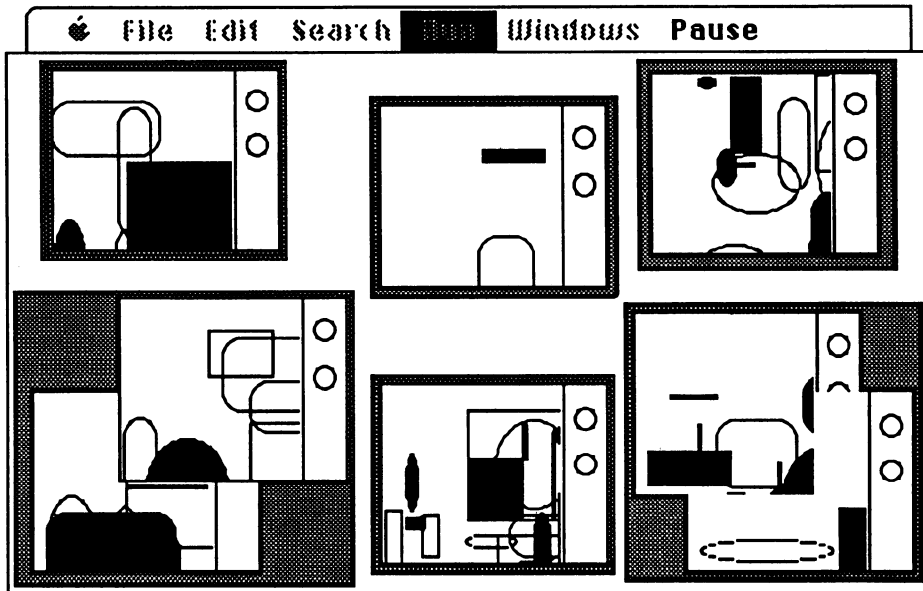


**Fig. 1 Ports\_Demo Output**

When one thinks of the Macintosh, one of the first things to come to mind is the Mac's frequent use of windows for multiple, independent areas or objects. Usually, one window is related to one task or concept, and tasks that have very different orientations are in separate windows.

There are several benefits for grouping items or tasks into separate windows. The most important is user interfacing. By using windows, the user can conceptually group similar or common things. This makes the use of software more intuitive because the user is shielded from modes in the software. In other words, instead of explicitly needing to have the computer switch to a different mode, all that is needed is to begin operations in a different window.

Obviously, the more intuitive for the user that your software is, the better. Indeed, one of the primary concepts of the Macintosh is that of using tools like icons, the mouse, and windows to decrease the need for special knowledge or special education. In this article, we will take a look at the concept behind windows, that of ports. (In a later article, we will examine windows more closely.)

In order to facilitate the creation of independent windows, the implementation of ports or graph ports provides for completely separate drawing environments. That is, each port is an entire drawing environment, and operations with one port are not related to another.

At any given time, after QuickDraw is initialized, a single port is always active, and is considered the current graph port. QuickDraw operations always occur within the current graph port.

With each graph port are associated several characteristics and parameters. These include location of the port's bit map, size, clipping region, visible region, background color, and several text mode values. The port's bit map refers to an area of memory where the bit image of any drawing is put. Usually, this coincides with an area of the screen ram, so that drawing is displayed on the screen. However, this is not a requirement. The port's bit map can be any area of memory on or off the screen. For example, a port may use an off-screen bit map to prepare an image, then move that image into the screen bit map.

The size of the port is determined by an enclosing rectangle. The rectangle encloses part of the bit map and indicates size and position within the bit map. For ports that are visible, the rectangle encloses an area of the screen.

Clipping region and visible region both restrict the available drawing area within a graph port. While the port itself is rectangular, the clipping and visible areas are regions, providing total flexibility with shape and configuration. The visible region acts like a clipping region, but is separate from it mainly for use by the window manager. (When one window overlaps another, the visible region of the bottom window is set to equal the part of the port that still shows.) There are two regions so that the user may have a region for clipping that is unrelated to the region needed by the window manager for handling overlap.

Each port is its own drawing environment, and as such has its own drawing characteristics such as pen size, pattern, position, etc. Also, each port has its own text drawing characteristics that are used by the font manager. This way, you need only choose which port in which to draw, and the font, style, and text size associated with that port are automatically used.

In addition to having its own locations, size, and modes, each graph port may also have its own coordinate system. That is, regardless of where on the screen the port is displayed, the

upper left corner of the port may be any coordinate. Usually, of course, the upper left coordinate is assigned the value 0,0. This is useful because neither the user nor the programmer need be concerned with the actual position of any port; drawing may be made as if the port were the entire universe. (You might begin to see why moving windows around is so easy; the coordinates inside the window remain constant.)

Since the available coordinate plane is much larger than any visible graph port, this feature allows ports to 'look at' different areas of the coordinate plane. For example, a program may draw on the entire plane, and different graph ports may be created as necessary to view separate parts of the entire area. This can be done without the need for conversions either in the drawing or the displaying; QuickDraw handles all conversions.

There are two important programming practices to develop with regard to ports. The first has already been mentioned: it is a good idea to group similar things, and separate dissimilar things. The second idea is that of port independence. Programs should be written to not rely on any port size or location. (When changing the size of a window, for instance, the program should NOT need a certain size window to perform.)

The sample program this month illustrates the second concept, as the first one is much easier to practice. In this program, the user 'creates' television sets on which drawing is displayed. While drawing is occurring, the user may create more TVs, or change which one is active. All this is done by clicking the mouse. If the mouse is clicked in a TV, it becomes the active one, if not, a new TV is created.

Each TV, of course, is a graph port. A port is created, and then in that port a television is drawn. Then, the clipping region of that port is restricted to the 'picture tube' part of the picture.

Drawing continues, regardless of which port is active, or where it is. The placement of the port, and the selection of the current port is entirely up to the user. Notice, that the main drawing procedure has absolutely no idea which port is active.

In order to avoid running out of memory, the number of possible ports is restricted. You may vary the constant 'MaxPorts' as you like. An array is set up called 'Screens' which contains pointers to the graph ports. QuickDraw procedures and functions that act upon ports take a pointer to the port as the parameter, not the port itself. The port is left to be dynamically allocated for flexibility. The type 'GrafPtr' is the predefined pointer to a graph port.

The variable ScrnCnt keeps track of how many ports we have created. Since we want to draw only on our TV screens, drawing does not occur if the user has not created any ports yet. When the user clicks the mouse, the program checks to see if the position of the mouse is inside any graph port. The point must first be brought to global coordinates, which makes it independent of any port. Then, one by one, the point is referenced to each graph port and checked against that port's rectangle. If the point is inside the rectangle, that port is selected as the current one.

When the point lies outside all ports, a new port is created. (Unless all possible ports have been created already; then the

program terminates.) The port is initialized and its characteristics are set. Then, the TV is drawn in the port, and the clipping region set. The new port is also set as the current port. Notice that the program explicitly sets the port's origin. Try playing with the origin values.

When the program ends, it closes each port one by one and frees the memory used by the ports. Then, it creates another temporary port that covers the entire screen. (The default values for the size and location of the port coincide with the screen.) This port is then filled with gray to eliminate the other graphics. Finally, this temporary port is also closed and its memory freed.

Although this is a simple example, it demonstrates a very powerful principle; that of independence. The more independent your programs are, the more flexible they are, and the more control the user may have. This avoids placing pointless limitations on your programs and their users. In addition, programming of this nature helps keep programs functional when the environment changes. (Like moving from a Mac to the Lisa with MacWorks, or future, improved machines.)

This concludes our introduction to the basics of QuickDraw. With this foundation, we can move on into the rest of the Macintosh with a strong background. You might want to try and improve on this sample program as an exercise to increase your skill. For example, try to allow the user to select the size as well as the location of the port. Or, try to prevent the ports from overlapping. If you're really enthusiastic, have all the TV sets going at once, and let the user change 'channels' on each of them, causing different graphics to appear on each channel.

In the next issue, we'll extend the discussion of ports to windows. We'll find out what the concept of windows adds to the already powerful idea of ports, and find out what features are provided to the user through the use of windows. Ciao.

#### **program Ports\_Demo;**

```
{ Ports_Demo - for MacTutor
-- by Chris Derossi}
```

#### **uses**

```
QuickDraw2; {QuickDraw2 contains the stuff for regions.}
```

#### **const**

```
MaxScreens = 8;
```

#### **type**

```
TVList = array[1..MaxScreens] of GrafPtr;
```

#### **var**

```
Screens : TVList;
ScrnCnt : INTEGER;
done : BOOLEAN;
```

#### **procedure SetUp;**

#### **begin**

```
HideAll;
```

```

ScrnCnt := 0;
done := FALSE;
end;

procedure ShutDown;

var
 TempPort : GrafPtr;

begin
 while ScrnCnt > 0 do
 begin
 ClosePort(Screens[ScrnCnt]);
 Dispose(Screens[ScrnCnt]);
 ScrnCnt := ScrnCnt - 1;
 end;

 { Create a new port and fill the screen with gray. }
 NEW(TempPort);
 OpenPort(TempPort);
 BackPat(Gray);
 EraseRect(TempPort^.PortRect);
 ClosePort(TempPort);
 Dispose(TempPort);
end;

function MakeTV (Left, Top : INTEGER) : GrafPtr;

var
 TempPtr : GrafPtr;
 WorkRect : Rect;

begin
 { Create a new port and set its characteristics. }
 NEW(TempPtr);
 OpenPort(TempPtr);
 PortSize(100, 80);
 MovePortTo(Left, Top);
 SetOrigin(0, 0);

 { Clear the port to white and draw our 'television' }
 BackPat(White);
 EraseRect(TempPtr^.PortRect);
 FrameRect(TempPtr^.PortRect);
 MoveTo(80, 0);
 LineTo(80, 80);
 SetRect(WorkRect, 85, 10, 95, 20);
 FrameOval(WorkRect);
 SetRect(WorkRect, 85, 30, 95, 40);
 FrameOval(WorkRect);

 { Restrict the clipping region to the TV 'screen' }
 WorkRect := TempPtr^.PortRect;
 WorkRect.right := WorkRect.Right - 20;
 InsetRect(WorkRect, 1, 1);
 ClipRect(WorkRect);
 MakeTV := TempPtr;
end;

procedure SetTV;

{ If the mouse is in one of our ports, select that port. If all}

```

```

{possible ports are created and the mouse is not in any port,}
{then we're done. Otherwise, create a new port.}

var
 MousePt, Pnt1 : Point;
 I, X, Y : INTEGER;
 TempPtr : GrafPtr;

begin
 TempPtr := nil;
 GetMouse(X, Y); { Current port's local coordinates }
 repeat { nothing }
 until not Button;

 { Convert to a point, the to global coordinates }
 MousePt.h := X;
 MousePt.v := Y;
 LocalToGlobal(MousePt);
 Pnt1 := MousePt;

 { Scan the existing ports. }
 if ScrnCnt > 0 then { we have some ports to scan }
 for I := 1 to ScrnCnt do
 begin
 SetPort(Screens[I]);
 MousePt := Pnt1;
 GlobalToLocal(MousePt);
 if PtInRect(MousePt, Screens[I]^PortRect) then
 TempPtr := Screens[I];
 end; { for loop }

 if TempPtr <> nil then { the mouse is in a port; set it }
 SetPort(TempPtr)
 else if ScrnCnt = MaxScreens then { all ports used }
 done := TRUE
 else
 begin
 ScrnCnt := ScrnCnt + 1;
 Screens[ScrnCnt] := MakeTV(Pnt1.h, Pnt1.v);
 end;
 end;

procedure MainLoop;

var
 Figure : INTEGER;
 WorkRect : Rect;

begin
 if Button then
 SetTV;

 if ScrnCnt > 0 then { do some drawing }
 begin
 if random mod 100 < 5 then { Clear the 'screen' }
 begin
 SetRect(WorkRect, 0, 0, 90, 90);
 EraseRect(WorkRect);
 end;

 { Create a random rectangle for drawing }
 WorkRect.top := random mod 80;

```

```
WorkRect.left := random mod 80;
WorkRect.right := WorkRect.left + (random mod 60);
WorkRect.bottom := WorkRect.top + (random mod 60);
Figure := random mod 6;
case Figure of
0 :
 FrameRect(WorkRect);
1 :
 FrameRoundRect(WorkRect, 18, 18);
2 :
 FrameOval(WorkRect);
3 :
 PaintRect(WorkRect);
4 :
```

```
 PaintRoundRect(WorkRect, 18, 18);
5 :
 PaintOval(WorkRect);
 end;
end;
end;

begin { Ports_Demo }
 SetUp;
 while not done do
 MainLoop;
 ShutDown;
 end.
```



# *Pascal Procedures*

## *Inline Traps Create Windows*



*Chris Derossi*  
*Editorial Board*  
*MacTutor Vol. 1 No. 5*

In the last issue of MacTutor, we introduced briefly the concept of graph ports and how they can make both programming and usage easier and more intuitive. However, the way that we used the ports, the programmer had to keep track of everything. It is fairly obvious that there will be several features that you, as a programmer, would want to associate with ports frequently, if not all the time. It is for this reason that windows exist.

Windows provide a front end to ports, making them more easy to use and providing you with several options and features. The Window Manager in the Macintosh Toolbox takes care of the tedious tasks for you. For example, in our last program we had to do everything with the port, including erasing it to white and drawing the border around the port. It is simple things like this which you need for every port, that the Window Manager handles.

In addition to taking care of the low level drawing, windows also provide you with more complex options. I'm sure you've seen the title bar on document windows, and the scroll bar controls are standard, too. The Window Manager also sets the window's coordinate system to local coordinates, with the point (0,0) at the upper left corner of the window. Also handled for you is the setting of the various regions. And, unlike ports, windows can handle overlapping and restoration of the window contents.

Let's back up a moment and take a look at the various different window kinds. There are four standard windows in the Macintosh, and you may create your own.

The four standard windows are shown in figure #1. Each type has an associated function on the Mac. Window type #0 is for documents, folders, and disks. Window type #1 is the normal dialog and alert box window, with the shadow border.

Type #2 is like type #1, but without a border. Type #16 is usually found as the window for a desk accessory. Types #0 and #16 have titles, while the other two do not.

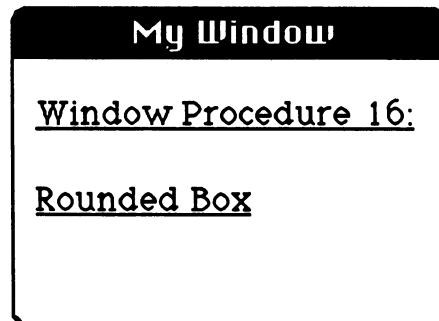
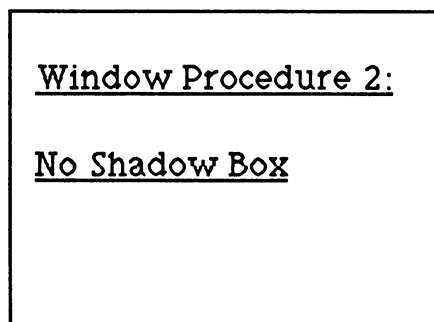
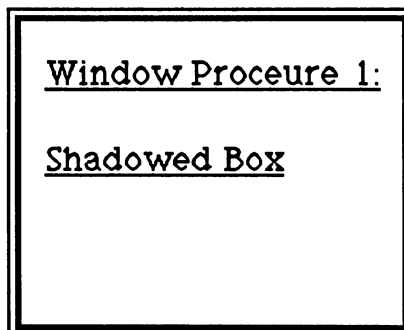
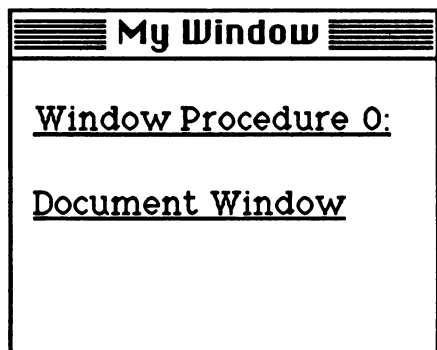


Fig. 1

For each type of window, the Window Manager handles drawing of the border and title bar. To do this, each window must have associated with it information about the type of window, and the title for that window. In fact, the magic about windows is just that: associating various things with a particular window.

Of course, the heart of the window is a GrafPort. Each window is its own port, and is unrelated to other windows. The rules for windows are the same as those for ports discussed last time. As you might have guessed, some of the other things associated with each window include window type and title string. There are also several status flags that are kept with each window that keep track of things like whether or not the window is visible, and if so, if it's highlighted.



Figure #2 shows a rough outline of a window record. Some of the fields are not shown, and many others are grouped under the heading 'Window Stuff'. We will worry about these at a later time. The very first thing in the record is the GrafPort. After that come most of the window flags and status variables. The windowDefProc is a handle to the procedure that draws the window. This is the procedure that handles the title bar and the border. The four standard windows each have a DefProc. You may create your own window definition procedure and have the Window Manager use it to draw the window.

The titleHandle is a handle to a string in memory. This is how the title is related to the window. Since controls are also associated with windows, a handle to a list of controls is also included in the record. More about this at a later time.

The next field is very important. The field nextWindow is a pointer to the next window record in a linked-list type data structure. By using this field, the Window Manager knows which window to highlight and make active if you get rid of the currently active one. Also, by using this scheme, the Window Manager needs to keep track of only one Window Record, and follow the pointer chain till it finds the one it wants.

Finally there is a field called refCon. This field has nothing at all to do with the window itself. Apple put that field there for your own use. You may have data associated with one of your windows, or a value associated with it. You may use this field as a pointer, handle, integer, or long integer. (Some type conversion may be needed, but you have enough bytes for any of those constructs.)

As with ports and other things, there are toolbox procedures and functions that create, destroy, modify, and control windows. The usual parameter passed to these routines is a WindowPtr, a pointer that points directly to the window record. In many cases, the Window Manager is looking for the data type GrafPtr instead of WindowPtr. This works because there is an entire GrafPort at the beginning of the window record.

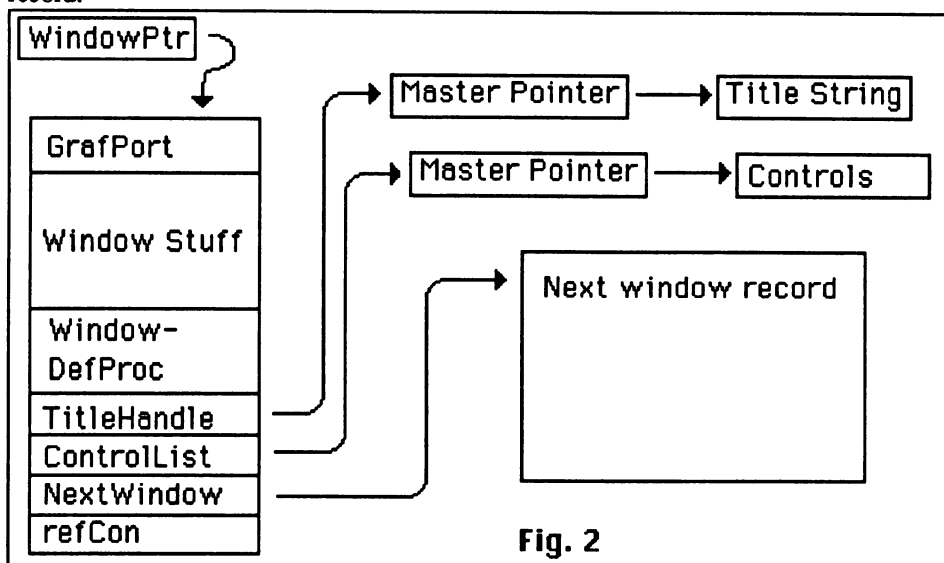


Fig. 2

## Inline Trap Calls From Pascal

We'll come back to windows in a moment; right now let's discuss the method for getting to toolbox routines that don't have a MacPascal interface. (Window routines are in this category.)

MacPascal has some built-in procedures and functions that call the Macintosh ROM, and even more routines in separate libraries. However, these routines fall very short of covering the entire set of toolbox utilities. For example, there is a toolbox function called 'OpenWindow', but there is no library available for MacPascal where this function is defined. Normally, there would be no way to interface to this and other similar routines, but realizing that doing so would be a common and important need, MacPascal has a short-cut set of routines known as InLines.

The InLine procedures and functions allow direct access to the Macintosh internals. Macintosh ROM routines are accessed by a thing called a trap vector. When the 68000 microprocessor comes upon an instruction that begins with a certain series of bits (1010, or \$A in hex), it looks up an address in a RAM table and jumps to it without question. The Mac ROM and the System program are responsible for setting up this table. It is through this method that routines in the ROM are accessed.

[NOTE: The trap mechanism has several advantages. First, as a programmer, you don't have to know the exact location of the routines you want to use. This allows the RAM table to change, and allows bugs to be repaired by replacing the ROM routine with a better one in RAM and updating the table. Also, you don't need to use the bytes necessary for a subroutine call. Instead, you need only the two-byte trap value. Lastly, the RAM table can be modified to create several nice effects. For example, you can intercept ALL of the traps to find out what's happening when debugging. Also, you can replace the ROM routines with your own routines that are better or different. All in all, trap vectors provide for a more versatile computer.]

The use of the InLine procedure call is fairly straightforward. The only mandatory parameter is the trap vector value. Each Mac ROM routine has a different trap vector value. You need to know what the trap vector is for the routine you wish to use. (Some of the window manager and quickdraw vectors are included this month. Look for a complete list in a later issue.)

## Do Your Own Type Checking!

Many of the procedures require parameters. You may simply pass

these parameters after the trap value. You may place as many parameters as you need in the call, and the parameters may be of ANY type. The InLine calls short circuit the normal type checking of MacPascal. For this reason, you should know exactly what you are doing when using an InLine call. If you pass a bad set of parameters, your Mac could crash and you could lose all data in memory and on disk!

There are two methods for passing parameters, by value and by reference. Simple data types (i.e. predefined) are passed by value unless they are VAR parameters. The simple data types include Integer, LongInt, Boolean, Char, Pointer, and Handle. All other data types are complex. Complex data types and VAR parameters are passed by reference. When a parameter is passed by value, the actual value of the parameter is placed on the stack. When passing by reference, a POINTER to the data is placed on the stack.

Since the InLine procedures are dumb, you must control what happens. Passed by value parameters can be passed normally, and complex types can be passed normally too as MacPascal knows to place a pointer on the stack. When you have a simple data type that is supposed to be a VAR parameter, you MUST force MacPascal to pass a pointer, not the data itself. To do this, use the @ function, which returns a pointer to the argument. For example, suppose that we have a variable MyNum which is of type integer. To pass the value of MyNum, you would pass MyNum itself. If you needed a VAR parameter, you would have to pass @MyNum.

The InLine call for procedures is InLineP(Trap, parm1, parm2, ...); You may have zero one or more parameters. There are three InLine calls for functions. Since InLine does not know what size value is to be passed back as the return value, a different call is provided for each type. When a value of size one byte is to be returned, use BInLineF(Trap, parms...). For two-byte long results, use WInLineF(Trap, parms...). W is for Word which is two bytes. L is for LongWord which is four bytes, which is the size of longints, pointers, and handles, so use LInLineF(Trap,parms...).

### Freedom From Large Libraries

Using InLine calls does not require any MacPascal libraries; InLine calls are built directly in to MacPascal. There is nothing to stop you from using InLine calls to the ROM even when equivalent routines already exist. In fact, there is a reason to do so. When you need any one procedure or function from a library, the entire library is loaded into memory. If you only need one function, most of the RAM being used by the other functions is wasted. By using a single InLine call instead of loading the entire library, you can save a significant amount of memory for your own data.

[NOTE: ANY single reference to the simple QuickDraw functions causes MacPascal to load the library QuickDraw1. In other words, even though you didn't have to do a 'uses QuickDraw1', it still happens, and you might consider replacing the calls with InLines.]

Now that you know how to access the Window Manager routines, let's look at the sample program. This month, the sample program is just a vehicle for the program's sole procedure, 'Message'. This procedure stands by itself, does not require any libraries, and may be ported to your own program. (With the exception of the type MsgArray, which must be declared before the procedure Message is.)

The procedure Message takes four inputs. First and second are the message you wish displayed. The first parameter is the number of lines of the array to use. The second is the array itself. Other methods could have been used, but we'll talk about that at the end. The next parameter is the window type, and can have values 0, 1, 2, or 16, corresponding to the window types in figure #1. Finally, a delay value is provided if you want to slow the output for effect.

### DISPOSE Should Be Disposed Of!

To create a window, it is necessary to provide the space for the window record. One way to do this would be to declare the entire window record type, but that is quite large. Instead, the size of the record is known to be 156 bytes. By declaring the record as an array of 78 integers, we achieve the same size data structure. Also, instead of using the built-in NEW procedure which allocates the memory, we create the variable ourselves. The reason for this is that the DISPOSE procedure that deallocates the memory doesn't work, and the memory never becomes free. Instead, we declare the variable, and its memory is freed when the procedure terminates.

### Avoid a Quickdraw1 Load

The InLine call to SetRect creates the window's rectangle. Notice that an InLine call is used to avoid loading QuickDraw1. Also, Rect is defined separately as well. (ANY reference to the data types in QuickDraw1 causes QuickDraw1 to be loaded.) Then, the InLine call for OpenWindow is executed. The Window Manager takes care of setting up the grafport, drawing the window, erasing it to white, and drawing the title bar if it has one. When this call is complete, the window will be on the screen.

Normally, this would cause the window to become the current grafport, but because MacPascal handles graphics and text in different windows, and therefore keeps changing the current port, it is a good idea to do a SetPort call. Once the window is set to be the current port, a simple loop calls DrawChar to place each character in the window.

When the loop terminates, the procedure waits for a mouse press and release. The call to the ROM routine Button is built into MacPascal, and no libraries need to be read. The procedure then closes the window and terminates. The Window Manager erases the window.

As an exercise, you may wish to modify the Message procedure directly. There is nothing to stop you from doing more elaborate drawing in the window. You might want to create your own alert procedures that draw bombs or other icons in the window. Additionally, you may want to pass the

window's title and rectangle to Message. You could also delete the delay if you don't want it.

That covers the basic introduction to windows. In the next issue, we'll talk about how to handle multiple windows, even overlapping ones. Also, we'll discuss the use of controls in general, and controls in windows. Up till now, we've limited ourselves to Button and GetMouse. The use of controls will greatly enhance our ability to receive input from the user. As you can see, windows with all their associated features provide you with a very powerful set of tools. Ciao.

You can produce your own error messages and alert boxes in your MacPascal programs.

Fig. 3

#### program Window\_Example;

```
{ This is the sample program for the
{ April issue of MacTutor Magazine.}
{ The main part of this program is only a
{ front end for the procedure}
{ called 'Message'. You can take the}
{ message procedure into your own}
{ programs as it stands.}
{ by -- Chris Derossi}
```

#### const

MaxLines = 5;

#### type

MsgArray =  
array[1..MaxLines] of Str255;

#### var

MyMsg : MsgArray;

#### procedure Message

(Lines : INTEGER;  
Msg : MsgArray;  
WindKind, Delay : INTEGER);

```
{ This procedure is the heart of this}
{ month's sample program. It will}
{ open a window on the desktop, display}
{ your message using the delay,}
{ and then wait for a press of the mouse}
{ button before it closes the}
{ window and returns.}
```

#### const

```
{Window Manager Traps}
NewWindow = $A913;
CloseWindow = $A92D;
```

```
{QuickDraw Traps}
SetRect = $A8A7;
SetPort = $A873;
MoveTo = $A893;
DrawChar = $A883;
```

#### type

Rect : array[1..4] of INTEGER;  
WindowRecord = array[1..78] of Integer;  
WindowPtr = ^WindowRecord;

#### var

Count, X, Index : INTEGER;  
Dummy : LongInt;  
TheWindow : WindowPtr;  
WindowStorage : WindowRecord;  
BoundsRect : Rect;

#### begin

```
{ Initialize memory, and open a new}
{ window. }
```

```
TheWindow := @WindowStorage;
InLineP(SetRect, @BoundsRect,
 40, 50, 390, 140);
Dummy := LInLineF(NewWindow,
 TheWindow,
 BoundsRect,
 'Message Window ',
 TRUE,
 WindKind,
 Pointer(-1),
 FALSE,
 0 + 0);
InLineP(SetPort, TheWindow);
```

```
{ Output the message, character by}
{ character. }
```

```
for Index := 1 to Lines do
begin
 X := Index * 15;
 InLineP(MoveTo, 5, X);
 If LENGTH(Msg[Index]) > 0 then
 for Count := 1 to LENGTH(Msg[Index]) do
 begin
 for X := 1 to Delay do
 ; {Nothing, just loop for a pause}
 InLineP(DrawChar, Msg[Index][Count]);
 end;
 end;
```

```
{ Wait for a button press and release.}
```

```
repeat
until Button;
repeat
until not Button;
```

```
{ Get rid of the window.}
```

```

InLineP(CloseWindow, TheWindow);
end;

begin {Window_Example}
HideAll;
MyMsg[1] := '';
MyMsg[2] := 'Your example program that
 demonstrates windows';
MyMsg[3] := 'has now begun.';
Message(3, MyMsg, 16, 0);

MyMsg[1] := 'You can produce your own
messages and alert';
MyMsg[2] := 'boxes in your MacPascal programs.';
Message(2, MyMsg, 1, 75);

```

error

```

MyMsg[1] := 'Pause and prompt for a
the mouse button.';
Message(1, MyMsg, 2, 75);

MyMsg[1] := 'You can also post brief
 instructions for your';
MyMsg[2] := 'users at any time while the
is running.';
Message(2, MyMsg, 0, 100);

MyMsg[1] := '';
MyMsg[2] := '';
MyMsg[3] := '';
MyMsg[4] := 'This demonstration program
over.';
Message(4, MyMsg, 16, 0);
end.

```

press of

program

is now



# Advanced Mac'ing

## Pascal's Generic Procedure



Steve Brecher  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 5

Macintosh Pascal provides a built-in procedure named **Generic** that is not documented. **Generic** permits you to call register-based Toolbox/OS routines. It can also be used to execute any machine language code that you have stored in a Pascal data structure. This article describes the use of the **Generic** procedure.

In effect, the following procedure is pre-declared:

```
procedure Generic(InstructionWord :
 integer;
 VAR Registers : RegRcd);
```

"RegRcd" denotes a data structure consisting of 13 32-bit values -- five address register values (A0..A4), followed by eight data register values (D0..D7). The exact type of Registers is immaterial, i.e., you could declare:

```
var
Registers : record
 A : array[0..4] of longint;
 D : array[0..7] of longint
end;
```

Or in some cases you might prefer something like:

```
var
Registers : record
 A0, A1, A2, A3, A4 : ^char;
 D0, D1, D2, D3, D4, D5, D6, D7 : longint
end;
```

The register values that you pass to **Generic** are written to the MC68000 registers. Then the one-word instruction denoted by the **InstructionWord** argument is executed. Finally, your Registers structure is updated with the (possibly) new values of the MC68000 registers before **Generic** returns to your program.

Any registers that are *not* used by the machine language routine you're invoking do not have to have their corresponding elements in the Registers data structure initialized. The machine language routine will have garbage in those registers (but, since it's not using them, it won't care).

Usually, **Generic** will be used to execute a register-based Toolbox/OS trap. In such cases the value you pass to **Generic** via the **InstructionWord** argument is the trap value. Here's an example routine that resets the modem output port to establish new communications parameters:

**type**

```
DataBitsT = (Five, Seven, Six, Eight);
 { ^--sic--^ }
ParityT = (OddParity, NoParity,
 EvenParity);
StopBitsT = (One, OnePointFive, Two);
```

```
function ResetSer (Baud : longint;
 DataBits : DataBitsT;
 Parity : ParityT;
 StopBits : StopBitsT) : boolean;
{returns true if no error, false if modem}
{port hasn't been opened yet}
{
```

**const**

```
PBControl = $A004; {trap value}
noErr = 0;
ModemOutRefNum = -7;
SerReset = 8; {CSCode}
```

}

**var**

```
ParamBlockRec : record
 Filler : array[0..11] of integer;
 ioRefNum : integer;
 csCode : integer;
 csParam : integer
end;
```

**Registers : record**

```
 A : array[0..4] of longint;
 D : array[0..7] of longint
end;
```

serConfig : longint;

**begin** {ResetSer}

**with** ParamBlockRec **do**

**begin**

```
ioRefNum := ModemOutRefNum;
csCode := SerReset;
serConfig := trunc(114571.7 / baud -
 1.338395)
```

```
 + 1024 * ord(DataBits)
 + 4096 * (ord(Parity) + 1)
 + 16384 * (ord(StopBits) + 1);
```

```
csParam := loword(serConfig);
```

**end;**

```
Registers.A[0] := ord(@ParamBlockRec);
```

```
Generic(PBControl, Registers);
```

```
ResetSer := (Registers.D[0] = noErr)
```

**end;** {ResetSer}

The **InstructionWord** argument to **Generic** does not have to be a trap value, however. It can be any 16-bit MC68000 instruction, as shown in the following example:

```

procedure CallCode(VAR Result :
 integer);
{}
{This example assumes a global integer }
{array named Code which contains a }
{machine language subroutine that takes}
{one VAR (address) argument on the }
{stack. CallCode calls the routine, }
{passing it the address of the Result }
{parameter. }
{}
const
 JsrlndirectA0 = $4E90; { Jsrl (A0) }
var
 Registers : record
 A : array[0..4] of ^integer;
 D : array[0..7] of longint
 end;
 Glue : array[1..4] of integer;

```

```

begin
 { MoveA.L (SP),A0 ;return addr }
 Glue[1] := $2057;
 { Move.L A2,(SP) ;ptr to Result }
 Glue[2] := $2E8A;
 { Move.L A0,-(SP) ;return addr }
 Glue[3] := $2F08;
 { Jmp (A1) ;to subr in Code array }
 Glue[4] := $4ED1;
 with Registers do
 begin
 A[0] := @Glue[1];
 A[1] := @Code[1];
 A[2] := @Result;
 end;
 {Call Glue routine, which invokes Code}
 {routine...}
 Generic(JsrlndirectA0, Registers);
 end; {CallCode}

```



# Pascal Procedures

## In Line Traps Simulate OS Calls



Alan Wootton  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 6

MacPascal is a miracle. Although billed as an educational product, and many serious programmers probably regard it as a toy (just as many consider the Mac to be merely a toy), MacPascal can be used to prototype professional software products.

Since MacPascal is not sold and supported as a development tool, a small amount of homework is needed to assemble a useful system. First you will need MacPascal. Second you will need RMaker, the resource compiler that runs on the Mac. For some an assembler is nice. And a 68000 debugger may be needed (I use MacDB by Bill Duvall). Since there is no syntax check on inline procedures, a debugger is useful to check the stack at the start of the call even if you are relatively bad at assembly language. Eventually one needs a compiler. For now all that is available is the Lisa Pascal Workshop. Hopefully, a Mac compiler will become available. For my purposes it is necessary for the compiler to generate *native code*; ruling out a p-code type of system (SoftTech Pascal won't do). Last but not least, *Inside Macintosh* is a must.

MacPascal provides methods to access almost all of the toolbox routines that make the Mac so great. The OS traps, however, are a bit tricky. As Steve Brecher showed us last month in MacTutor with his Advanced Mac'ing column, the undocumented GENERIC procedure is used to call OS traps. However, INLINE can also be used for OS traps, as we shall see this month in this example of using STUFFHEX from Pascal. (See Steve's Advanced Mac'ing column in this month's issue for an example of using STUFFHEX from Basic.)

### Two Types of Toolbox Traps

There are two types of toolbox trap; those that are stack based and those that are register based. The stack based calls are supported using the INLINE procedure. The register based traps, of which most of the OS traps are typical, are more difficult to access because there is no way (other than using GENERIC) to set registers from Pascal. Fortunately, virtually all of the register traps use only D0 and A0. Therefore, if we could set D0 and A0 before the trap and read them afterwards, we would have an OS trap call mechanism using the INLINE procedure. I have written a short piece of 68000 code to accomplish this.

As an example, I have prepared a short program to read the date and time from the system. Two calls are made to the toolbox to do this; both are register based. Note that when a pointer is to be passed in A0, the @ operator must be used to set the variable to the correct value before the call. Also note that for any \$A0xx calls that return a value to the calling routine, the flag \$100 should be added to *not* preserve all registers.

### Glue Routine for Pascal to OS

Our machine language routine will pop from the stack, the addresses of the variables which we wish to place in the registers A0 and D0. We will pop these addresses off the stack and use them to load the variables into the two registers. The new values returned in A0 and D0 will then be copied back into those same two Pascal variables. Note that after the inlineP call, all registers are saved so we can use them any way we please. To return to Pascal, we are using an RTS instruction so it will be necessary for our machine language routine to calculate our return address. Fortunately A0 points at the word before where we must return.

#### ROUTINE TO LOAD A0 AND D0 FOR OS TRAP CALLS FROM MACPASCAL.

```

126: 2848 MOVE.L A0,A4 ;SAVE RETURN
128: 548C ADDQ.L #$2,A4 ;POINT TO RET
;
; get address where trap will go into A0
;
12A: 41FA 000C LEA *+$000E,A0
;
; pop trap from stack and place into code
;
12E: 309F MOVE.W (A7)+,(A0)
;
; get A0 and D0 from stack
;
130: 245F MOVE.L (A7)+,A2 ;POP A0 VAR
132: 265F MOVE.L (A7)+,A3 ;POP D0 VAR
134: 2052 MOVE.L (A2),A0 ;LOAD A0
136: 2013 MOVE.L (A3),D0 ;LOAD D0
;
; this word gets overwritten by real trap
;
138: FFFF .word $FFFF ;do trap now
;
; pass back A0 and D0 contents
;
13A: 2488 MOVE.L A0,(A2) ;SAVE RET A0
13C: 2680 MOVE.L D0,(A3) ;SAVE RET D0
13E: 4ED4 JMP (A4) ;RETURN

```

We will stuff the machine code for our little OS trap caller above into memory by using the STUFFHEX routine from our Pascal program. The above listing is provided to show how the machine language patch works. Two OS traps, ReadDateTime and Secs2Date will be called using our OS trap caller, returning the information in variables of our DateTime

record. From this, it should be obvious how to access other OS type traps using INLINE procedures. (As mentioned, the undocumented GENERIC can also be used for this purpose.)

```
program show_date_time;
```

```
{ This is a test of a method of accessing }
{ the Mac OS toolbox traps from Mac }
{ Pascal that are register based, using }
{ A0 and D0 only. The call will be: }
{ inlineP($4E75,@d0, @a0, trap,@access)}
{ d0 and a0 hold values of registers D0 }
{ and A0 before and after the call. Trap }
{ refers to the trap address being called.}
```

```
type
 DaTimRec = record
 year: integer;
 month: integer;
 day: integer;
 hour: integer;
 minute: integer;
 second: integer;
 dayofweek: integer;
 end;
```

```
var
 currentTime: DatimRec;
```

```
procedure GetTime(var date: DaTimRec);
var
 access: array[0..12] of integer;
```

```
d0,a0, secs: longint;
```

```
begin
 stuffhex(@access,'2848548C41FA000C309F245F265F2
0522013FFFFF248826804ED4');
 a0:=ord(@secs);
```

```
inlineP($4E75,@d0,@a0,$A139, @access);
{ReadDateTime, pointer to secs in A0}
```

```
a0:=ord(@date);
```

```
inlineP($4E75,@secs,@a0,$A9C6,@access);
{Secs2Date, secs in d0, pointer to record}
{is in A0.}
```

```
end;
begin {main}
 GetTime(currentTime);
 with currentTime do
 begin
 writeln('year is', year);
 writeln('month is', month);
 writeln('day is', day);
 writeln('hour is', hour);
 writeln('minute is', minute);
 writeln('second is', second);
 writeln('year is', year);
 writeln('day of week is', dayofweek);
 end;
```

```
end.
```





# Advanced Mac'ing

## I/O Completion, Stuffhex and Miss Elaine E.



Steve Brecher  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 6

### I/O Completion Routines

Application I/O requests are ultimately performed by low-level operating system trap routines: `_Read`, `_Write`, `_Control`, `_Status`, `_Open`, `_Close`. The program passes the address of an I/O parameter block to the trap routine. The parameter block specifies the device driver (e.g., Sony floppy driver, serial driver) and the particulars of the request such as (for `_Read` or `_Write`) the number of bytes and the address of the program's buffer.

The operating system passes the parameter block on to the device driver; or, if the driver is currently busy, the parameter block is put onto the driver's queue of pending work.

Mac I/O operations can be either synchronous or asynchronous. A synchronous operation will be completed before control is returned to the invoking program. An asynchronous operation will be initiated - started upon by the driver, or put into its queue - and then control will be returned to the program regardless of whether the operation is yet complete.

Asynchronous I/O is the heart of multi-tasking; indeed, the only rationale for multi-tasking is the overlapping of I/O and computation. Programs can implement a primitive form of multi-tasking within themselves by using a multi-buffer I/O scheme. For instance, a program can request that a buffer full of data be written, and then - before that write request is completed - immediately start filling another buffer with data.

Completion routines can be a convenient mechanism for a program to manage multi-buffered I/O. The completion routine is called asynchronously with respect to the rest of the program; it can mark the buffer associated with the just-completed I/O request as free (written) or ready for processing (read). If another buffer is ready to be read into or written from, the completion routine can initiate the next I/O request.

Completion routines can also be used to hide low-level I/O details from the mainline logic of a program. I attempted to use the Mac's completion routine facility for that purpose in a program doing input from the modem port at high baud rates. The program started off by issuing an asynchronous read request for one byte, and specifying a completion routine. The completion routine looked like this:

1. Put the received byte into program's input buffer;
2. Issue a `_Status` call to find out whether the serial input driver has more data in its own buffer;
3. If the driver has more data available, issue a `_Read` to

transfer it to the program's input buffer;

4. Re-issue the one-byte asynchronous `_Read`.

The idea was for the completion routine to handle the low-level details of managing input into the program's buffer. The mainline of the program would then be freed from concern with those details, and could just take data from the buffer as it became available.

But this scheme didn't work. Following is an explanation of why it didn't, which relates to a severe limitation in the Mac's implementation of completion routines.

Asynchronous I/O is requested by setting bit 10 (\$0400) in the trap value. There are two ways for a program to detect the completion of asynchronous I/O. When an I/O operation is initiated, the `ioResult` field of the parameter block is set to 1; when the operation completes, `ioResult` is set to 0 (if no error) or to a negative error code. So the program may test `ioResult` to determine whether the operation is complete. Or, the program may specify the address of an I/O completion routine in the `ioCompletion` field of the parameter block. If `ioCompletion` is not nil (zero), then when the operation is complete, the OS will JSR to the address contained in `ioCompletion`.

If bit 10 of the trap word is clear - a synchronous I/O - the Device Manager will clear `ioCompletion` before initiating the I/O; and the OS will test `ioResult` in a tight loop, waiting for it to become zero or negative before returning from the trap. Hence, you don't need to clear `ioCompletion` prior to a synchronous I/O; but for asynchronous I/O you must either clear `ioCompletion` or set it to the address of a completion routine.

When a completion routine is entered, register D0.W contains the `ioResult` value (and the condition codes reflect a TST of its value); A0 contains the parameter block address; 4(SP) - just above the return address - contains a pointer to the driver's Device Control Entry (DCE); and registers D0-D3/A0-A3 may be altered by the completion routine. The I/O queue element (a.k.a. the parameter block) will have been unlinked from the driver's queue. And - here's the catch - interrupts may or may not be disabled.

The OS initiates an I/O operation by calling the appropriate entry point of the device driver. Let's consider, for example, a read request to the serial input driver. The OS calls the serial input driver's "Prime" (read) routine. There are two possibilities: either the request will be completed immediately by the Prime routine, or it will be completed later as the result of an interrupt.

If the number of characters in the driver's buffer is equal to or greater than the number specified in the I/O request, the Prime routine will fulfill the request immediately by transferring characters to the caller's `ioBuffer`, and then jump to the OS `ioDone` routine, which will call the completion routine. Interrupts are enabled at entry to the completion routine, and it can therefore do anything that any other part of an application can do. In this case, when the completion routine is entered the original `_Read` trap has not yet returned and no other application code has been executed. Therefore the effect is similar to that of the application executing the following:

```
_Read ;synchronous
JSR CompletionRoutine
```

If the completion routine initiates another I/O request to the same driver, specifying the same `ioCompletion` (i.e., itself), that request may also complete immediately, in which case the completion routine will be re-entered.

If there are insufficient characters in the driver's buffer to satisfy a read request, the Prime routine will return to the OS, which will return from the trap. Later, the driver's received-character interrupt service will determine that the request is complete, and jump to the OS `ioDone` routine, which will call the completion routine. Now, however, the completion routine is entered with interrupts disabled; in effect the completion routine is part of the interrupt service.

The completion routine can do no significant processing, because that would hold off interrupts for too long a time - for example, subsequent incoming serial data might be lost. This virtually excludes the possibility of initiating a new I/O operation from within the completion routine. Also, the completion routine (at interrupt level) cannot do anything that would alter the heap configuration: the interrupt might have occurred while a handle was being dereferenced.

Since there is no point to doing asynchronous I/O unless it is assumed that the request will not complete immediately, the programmer must assume that a serial I/O completion routine will be executed at interrupt level. This makes the completion routine facility rather useless for serial I/O. The routine could set a flag indicating the completion, but such a flag is already available in `ioResult`.

Wish list item... To make completion routines useful, the Mac OS would have to be enhanced to implement what the DEC PDP-11 operating systems refer to as "fork processes." Fork processes are serialized and executed synchronously after all the (possibly nested) interrupts have been dismissed, but before control is returned to the point at which the first interrupt occurred. This enables completion routines to be interruptable, giving them time to do useful work.

### MS BASIC Programmers: Stuff It!

The QuickDraw `StuffHex` routine is a fast way to

convert lengthy hex machine language data to binary code in MS BASIC programs. The trick is to prefix each string of hex digits with a character whose ASCII value is equal to the number of hex digits: this character is thus a length byte, making the string a Pascal string which can be passed to `StuffHex`. Note that the address of the string data is not given by the address of the BASIC string variable. The string variable is actually a data structure containing information *about* the string, including its address. The address can change each time the string is assigned or read into. We get the address of the string data by `PEEK`ing into the string variable. Example:

```
' Machine language interface to StuffHex:
'
DIM SH$(2)
SH$(0)=&H245F : SH$(1)=&HA866
SH$(2)=&H4ED2
' Array that gets the binary machine
' language:
DIM CODE$(SomeAdequateSize)
' Declare all scalars before getting array
' addresses:
CODEPTR!=0! : HEXLINE$="" : STRINGPTR!=0!
P!=0!
' Get addresses of arrays and the string
' variable:
STUFFHEX!=VARPTR(SH$(0))
CODEPTR!=VARPTR(CODE$(0))
P!=VARPTR(HEXLINE$)
' Read lines of hex data, convert to binary:
READ HEXLINE$
WHILE HEXLINE$<>" "
' Get address of first byte of string data
' (the length byte):
STRINGPTR!=(PEEK(P!+2!)*65536!)+
 (PEEK(P!+3!)*256!)+PEEK(P!+4!)
' Convert the line's hex data to binary:
CALL STUFFHEX!(CODEPTR!,STRINGPTR!)
' Adjust pointer into CODE! array for next
' data (if any):
CODEPTR!=CODEPTR!+(ASC(HEXLINE$)/2)
READ HEXLINE$
WEND
'
' The ASCII value of the first character
' of each string must be equal to the
' number of characters in the rest of the
' string. Thus the number of hex digits
' in each string must be at least 32
' (ASCII space) and no greater than
' 126 (ASCII ~) so that the length is a
' displayable ASCII character. But the
' length must be other than 34 -- 34 is
' the ASCII code for quotation mark!
' Note the following DATA line has 36 hex
' digits, and that the first character of the
' string is "$", i.e., CHR$(36).
DATA "$0123456789ABCDEF0123456789ABCDEF0123"
'
' More lines of hex data would go here
```

' Empty string marks end of data:

DATA ""

### Reports from Miss Elaine E.

Due to a **QuickDraw bug**, **DrawText** in **srcCopy** mode will erase four or five character positions after the position of the last character in the string. It's OK to use **srcXor** if no character position will be overwritten; but if you need **srcCopy**, a workaround is to set the right side of the port's **clipRegion** to the right edge of the last character before calling **DrawText**. If the font is not monospace, this implies calling **TextWidth** first to find out the screen width of the string to be drawn.

The **QuickDraw ScrollRect** routine will be slowed by a factor of 3 or 4 if you include the borders of the **GrafPort** in the rect that is scrolled. Make sure the rect passed to **ScrollRect** is inset at least a couple of pixels from the port's border(s).

Thanks to Steve Hanna for this tip on the **alternate screen buffer**... To have **QuickDraw** draw in the alternate screen buffer, change the **BaseAddr** field of the **GrafPort** to \$72700. When you use QD calls with that **GrafPort** as the current port, the drawing will be done in the alternate screen buffer. To flip the display between the main and alternate screen buffers, complement bit 6 of **VIA** buffer A, which is mapped to address \$EFFFFE. (Steve Hanna also notes that the low-order 3 bits of buffer A control sound volume [0..7]. Bit 7 is tied to the **SCC** -- see below). The following instruction will flip the display to the other screen buffer:

Bchg.B #6,\$EFFFFE

Thanks to Dennis Brothers for pointing out that the **MacsBug HD** (heap display) command is a way to find your program in memory - the first **CODE** resource in the heap is most likely the first (or only) segment of your program.

The reason the **ROM serial driver** doesn't support input flow control is that a **D0** was coded where a **D3** should have been; a two-bit error. And the reason mouse interrupts will be lost if you close the **ROM serial driver** (without immediately opening the **RAM serial driver**) is that the **ROM serial driver** neglects to finish its cleanup by setting the master interrupt enable bit in the **SCC** chip - a one-word omission from a table. (Mouse movement signals come in through the **DCD** pins of the **SCC** and generate **SCC** external status interrupts.)

If you want to program the **SCC** yourself, ask your local **Zilog** office for a copy of the "**Z8030/Z8530 SCC Serial Communications Controller Technical Manual**." The memory-mapped addresses of the **SCC** are in the **MDS** equate file **SysEquates.Txt**. The **SCC WAIT/REQUEST** (asserted low) pin state is brought over to bit 7 of **VIA** (6522 chip) buffer A, which is mapped to memory address \$EFFFFE. The serial driver's **Open** routine configures the **SCC** to assert **WAIT/REQUEST** when an input character is available in the

**SCC** buffer. This enables the Sony floppy driver to feed incoming data to the serial driver's input buffer while the Sony driver has disabled interrupts. When the byte at \$EFFFFE is positive (bit 7 clear), the Sony driver fetches the character waiting in the **SCC** and calls code in the serial driver interrupt service routine.

Inside Macintosh says of the serial status **ctsHold** flag (**Serial Driver**, p. 13), "If output has been suspended because the hardware handshake has been negated, **ctsHold** will be nonzero." The conditional clause could be misleading: provided that the output driver has been opened, **ctsHold** always reflects the state of the **CTS** (asserted low) pin of the **SCC** (pin 7 of the **DB-9** connector) regardless of the status (or existence) of any output request. If **CTS** is asserted, **ctsHold** is zero.

Need a quick (and dirty!) test of whether any **OS events** are pending? Address \$014C contains the **OS** event queue header (a pointer); if it's nil (zero), the queue is empty.

Think you have a good (homemade) backup of **Macintosh Pascal**? Make sure you can click the mouse 101 times in the source edit window. On the 101st click, **Mac Pascal** goes to the disk to check that the master is there; if it doesn't like what it finds, it abruptly quits to the **Finder** (trashing any of your unsaved work). In my book, this qualifies as a "worm" and, since it's not documented, is tasteless at best and unethical at worst. If a publisher wants to frustrate users who attempt to backup a product or use it on a hard disk without inserting the master, the program should either quit at the outset or put up a dialog box demanding the master. The more experienced the programmer, the more violent his aversion to being forced to use distribution media in production work.

**Consulair Mac C** users who want all string constants to be compiled in **Pascal** format (length-byte prefix) can include the following at near the top of the source file:

```
#asm
 STRING_FORMAT 3
#endasm
```

If there's no semantic difference between **pre-incrementing** and **post-incrementing** a variable in your **Mac C** program (i.e., you can choose either **++i** or **i++**), use **pre-increment**: it generates more efficient code. Same applies to decrements. If you use **post-inc(dec)rement**, the generated code will **inc(dec)rement** the variable in memory, then offset that operation by **dec(inc)rementing** it in a register in preparation for its previous value being used in an expression, even if it's not so used.

A **Mac C** update is expected to be available this spring with floating point, register variables, structure assignments, and slicker code generation. Look for the **Greenhills C** compiler under the Apple name on the Mac later this year: it's being ported from the Lisa along with the **Workshop**. I haven't used it, but it's reported to generate slick code.



# *Pascal Procedures*

## *Reading Paint Files from Pascal*



Alan Wootton  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 7

Here's a program to view MacPaint documents from MacPascal (like "showpage"). I use the File Manager, the Standard File Package, the toolbox procedure UnPackBits and QuickDraw's CopyBits to display the image loaded in memory.

In MacTutor #5, Gary Palmer asked about loading MacPaint files in MacPascal. Well, I remembered once running across a description of MacPaint docs by Bill Atkinson. After some digging, I found the description in part of a mailing from tech support to developers dated Dec. 1983.

Atkinson explains that the resource fork is not used and that a 512 byte header precedes the *packed* bitmap data. The header is an array of patterns and some empty space. To simply view the picture we can ignore the header. The MacPaint 'page' is 720 lines of 72 bytes each. This is almost 51K and since we wish to load this into memory, a FatMac will probably be required. Atkinson gives a short routine to decode the data into a page. He uses UnpackBits, which is on page 7 of ToolBox Utilities.

Reading the data was a problem. I tried to use the file I/O routines provided with MacPascal. One must either read data byte-at-a-time, or, if you read large blocks, you can never get the last partially filled block. At this point I decided that Real Programmers use the low level File Manager routines. In other words PBOpen, PBRead, and then PBClose.

Another MacPascal shortcoming is the GetOldFile function. Once again, Real Programmers go straight past the limitations of the language and consult the bible (Inside Mac). We shall use the Standard File Package.

### **The Standard File Package**

MacPascal's function GetOldName will only select files of type TEXT. To select MacPaint docs it will be necessary to call the Standard File Package directly (OldPaintName in following program). On page 30 of Packages in "Inside Macintosh", the procedure SFGetFile is described. To call it, use trap \$A9EA, which is Pack3, and be sure to push a 2, the 'selector', on the stack last. You must type in the SFReply record on page 25. [For a complete description of SFReply, see the Assembly Column in this issue.] Since our type list will just be one type (PNTG) it will not be necessary to declare an array. Pass nil for filefilter and dlhook. There is an argument to pass a prompt string that Inside Mac says is historical only. I find that it works fine.

### **Low Level File Manager**

The filecall interface in the following program simulates the PB calls of the File Manager. This is an example of the OS call I presented in MacTutor #6. [Please refer to issue #6

for details on calling OS routines from Pascal.] Before you can use the the PB routines, you must type the lengthy Parameter Block declarations. There are four variant parts to this record. I have only used the ioParam part here because that is all that was needed in this example. You should type the whole thing and save it somewhere for later use. Note that 8 bit types do not come out right, so one must take care that the declaration has the correct length. The routine descriptions (starting on page 31 of the File Manager) give a nice list of those parameters that must be set before calling (assembly programmers note: in some places ioRefNum is listed as 22, but the correct number is 24). You should carefully check that each parameter is set correctly before calling. The exception is ioCompletion, which can be ignored since we will be making synchronous calls. If ioNamePtr is not nil then the File Manager assumes that its value is the address of a string. This could badly mess up memory. In MacPascal, declared variables are cleared to 0's so you can get away with this, but later, when you compile, nasty bugs pop up (this actually happened to me).

### **UnPackBits**

UnPackBits is actually very simple. I complicated things by not reading all the data into memory at once. The scheme used is to read 1024 bytes into memory (skipping the header) and when the first 512 bytes are used I slide the upper 512 bytes down and add 512 bytes onto the end. The program unpacks 72 bytes at a time for 720 lines. For faster operation, unpack 4 lines at a time. Change 720 to 180 and 72 to 288.

After the page array is filled, it would be nice to take a look at it. Copybits is used to move the image onto the drawing window. The nice thing about copybits is that it will scale the image to fit. Simply setup a Quickdraw Bitmap record for page, set destrect to the desired size, and call copybits as shown. If you change 144 to 576 and 180 to 720 you can see the painting full size.

To write a MacPaint document (not done in this program) it is necessary to know the format of the header. The header is a 4 byte version number (default = 2) and then  $38 \times 8 = 304$  bytes of patterns and then 204 unused bytes for a total of 512. However, Atkinson's example writes 512 zeros for a header so I guess that would work. To pack the data you use the reverse of the unpack procedure. Things can be made simpler though because you can use one destination buffer and then write it out after every PackBits.

It is my contention that everything in Inside Mac can be demonstrated and used from MacPascal (except things like Vertical Retrace routines). If there is anything that you have trouble with and would like to see in MacTutor, write a letter to MacTutor and I will see if there isn't a way to do it.

```

program Read_MacPaint_doc; { by Alan Wootton 4/85 }
 { This program reads a MacPaint file and shows
 the picture 1/4 size in Drawing }

uses
 Quickdraw2;
type
 ptr = ^integer;
 OType = longint;

 SFRReply = record { this is used by the standard file
 package, see Packages }
 good : integer;
 ftype : OType;
 vrefNum : integer;
 version : integer;
 fname : string[63];
 end;

 { Parameter Block information contained in File Manager
 chapter of Inside Macintosh. Note that MacPascal won't do 8
 bit fields right }

 ParamBlkPtr = ^ParamBlockRec;
 ParamBlockRec = record
 { data structure of File Manager }
 qLink : Ptr;
 qType : integer;
 ioTrap : integer;
 ioCmdAddr : ptr;
 ioCompletion : ptr;
 ioResult : integer;
 ioNamePtr : ^str255;
 ioVrefNum : integer;
 { case ParamBlkType of ... ioParam: }
 ioRefNum : integer;
 { ioVersNum : byte; }
 ioPermsn : integer; { byte }
 ioMisc : ptr;
 ioBuffer : ptr;
 ioReqCount : longint;
 ioActCount : longint;
 ioPosMode : integer;
 ioPosOffset : longint;
 end;

 diskBlock = packed array[0..511] of char;

 paintPage = array[0..35, 0..719] of integer;
 { 576*720 bitmap = 72 bytes * 720 lines = 51,840 bytes }

var
 reply : SFRReply; { used by OldPaintName, declared here
 to access vRefNum }
 filename : str255;
 fileblock : ParamBlockRec;
 buff : array[0..1] of diskblock; { 1K }

 Page : paintPage; { Huge variable, almost 52K }

```

```

 srcPnt : ^diskBlock;
 destPnt : ^paintPage;

 lines, err : integer;

 curport : Grafptr;
 pageBits : bitmap;
 destRect : Rect;

 { common OS trap code, could be done with 'Generic' call }
 { see MacTutor Vol. 1 No. 6 for assembly source code }
 function filecall (Pb : ParamBlkPtr;
 trap : integer) : integer;
 var
 access : array[0..12] of integer;
 d0, a0 : longint;
 begin
 stuffHex(@access,
'2848548C41FA000C309F245F265F20522013FFFF224826804
ED4');
 a0 := ord(pb);
 inlineP($4E75, @d0, @a0, trap, @access);
 filecall := loword(d0);
 end;

 { The following File Manager calls work just like }
 { those described in Inside Macintosh for the Lisa Pascal
 Workshop. }
 { Except that the async parameter is a dummy; all calls are
 sync }

 function PBOpen (Pb : ParamBlkPtr;
 async : boolean) : integer;
 begin
 PBOpen := filecall(pb, $A000);
 end;

 function PBClose (Pb : ParamBlkPtr;
 async : boolean) : integer;
 begin
 PBClose := filecall(pb, $A001);
 end;

 function PBRead (Pb : ParamBlkPtr;
 async : boolean) : integer;
 begin
 PBRead := filecall(pb, $A002);
 end;

 { same as OldFileName, only for MacPaint docs }
 function OldPaintName (prompt : str255) : str255;
 var
 where : point;
 typelist : longint;
 begin
 reply.good := 0;
 typelist := $504E5447; { 'PNTG' }
 setPT(where, 100, 100);
 inlineP($A9EA, where, @prompt, nil, 1, @typelist, nil,

```

```

@reply, 2);
{ $A9EA is pack3 , 2 is SFGGetFile, see Packages }
if reply.good < 0 then
 OldPaintName := reply.Fname
else
 OldPaintName := "";
end;

{ This procedure moves buff[1] down onto buff[0], adjusts
srcPnt to point at it's same data and tacks more data on end in
buff[1]. }
procedure Refill_Buffer;
begin
 blockmove(@buff[1], @buff[0], 512);
 srcPnt := pointer(ord(srcPnt) - 512);
 fileBlock.ioBuffer := @buff[1];
 fileBlock.ioReqCount := 512;
 fileBlock.ioPosMode := 0;
 err := PBRead(@fileBlock, false);{ fill buff[1] }
end;

begin { main }
ShowDrawing;
fileName := OldPaintName(' Select Painting to examine');
while filename <> " do
 begin
 fileBlock.ioVrefNum := reply.vrefnum;
 fileBlock.ioNamePtr := @fileName;
 fileBlock.ioPermsn := 1;{ read only, version number
 is in upper byte and is 0 }

 fileBlock.ioMisc := nil;
 if PBOpen(@fileBlock, false) = 0 then
 begin
 refill_buffer;{ first block is skipped }
 refill_buffer;{ fill 1 }
 refill_buffer;{ 1 onto 0, refill 1 }

 srcPnt := @buff[0];{ <--- pointer to start of data }

```

```

destPnt := @Page;{ <---- pointer to Page bitmap }

{-----> This is the part that actually decodes the packed
MacPaint data <-----}
{ $A8D0 is UnPackBits, see page 7 in Toolbox Utilities }
{ srcPnt and destPnt are updated by UnPackBits so use @}
for lines := 1 to 720 do{ 720 lines in paint doc }
 begin
 inlineP($A8D0, @srcPnt, @destPnt, 72);
 { UnPackBits for 72 bytes }
 if ord(srcpnt) >= ord(@buff[1]) then
 Refill_Buffer; { if src no longer in buff[0] }
 end;{ of 720 lines }

if PBClose(@fileBlock, false) < 0 then
 writeln('close error');

{ now show drawing in current port (drawing window) }
 GetPort(curPort);

{ set up bitmap record }
 pagebits.baseaddr := pointer(ord(@page));
 pagebits.rowbytes := 72;
 setRect(pagebits.bounds, 0, 0, 576, 720);

 setRect(destRect, 0, 0, 144, 180);
{ 144,180 =1/4 normal, image will be squashed to fit }

 copyBits(pagebits, curport^.portbits, pagebits.bounds,
 destRect, srcCopy, nil);
 moveto(4, 190);
 drawstring('Click to open another MacPainting');
 repeat
 until button;

 end;{ if no Open error }
 fileName := OldPaintName(' Select another Painting ?');
 end;{ if name<>" loop }
end.

```



# *Pascal Procedures*

## *Custom Dialog Box for Input*



Alan Wootton  
*MacTutor Contributing Editor*  
*MacTutor Vol. 1 No. 8*

Our Project for this month is to create a pair of input statements for MacPascal. In Basic there's a command that prompts the user for a number or a string. It would be nice to have this for MacPascal, but would be even better if it could be implemented using the excellent Macintosh user interface. Towards this end, we will be using the Dialog Manager, Package 7, and Resources.

### **Resources**

Previous generations of computers provided only files as storage entities. Any small or large chunk of numbers to be accessed by the operating system usually required a file to hold it. If a program needed a large number of small data areas, the result was usually a large number of files. This happened all too often.

The Mac is different. On the Mac a file name actually represents two files. One of these is the usual type of file, the other is full of tiny subfiles. A whole new world! Naturally, there is a sort of directory to go with these subfiles.

*Inside Mac* calls the two files "forks" of a file. The subfiles are "resources" and the directory is the "resource map". In a normal file system all files are referred to by name and possibly a version number. Resources are referred to by a four character name called the "TYPE" (actually any longint) and an integer known as the "ID number". Many of the ROM routines access a particular TYPE with the ID being variable. For instance, a menu can be created by providing a number and the Menu Manager will access the resource with TYPE='MENU' and the ID provided.

File access can be quite a programming hassle, so you are probably imagining that resource access is even worse. Not true. There is no way to read in only part of a resource, so the subtleties of file I/O can be ignored. You provide a name and the resource manager will open the file for you. When you provide a TYPE and an ID the resource manager will search the maps that are open, read the data in, if necessary, and return a handle to that data. There are some fine points concerning control over when the data is read in, when it is to be purged from memory, which heap it goes on, and others. For our purposes it is enough to just open the file.

### **Resource Tools**

Every computer has some sort of software to manipulate and view the files, so what does one use for resources? There are four tools from Apple to work with resources.

The simplest is RMover. RMover will simply list the resources in a file. You may also cut and paste resources

between files. RMover will recognize some types as probably having a certain form of data and display it appropriately. Examples are STR (a string), PICT (a picture), ICON (an icon). In each case the data is simply so many bytes, but it is assumed that a particular type is used exclusively for a single brand of data (which is not a law). Every Mac freak should rummage through the files on his disks with RMover to get a feel for what is there (check out MacPascal!).

For programming, the next most useful tool is RMaker. This is a sort of compiler for resources. It knows the form of the data for windows, menus, dialog boxes, and more. Make a text file with coded descriptions in it, run RMaker and a new file is created with the described resources in it. RMaker is supplied with the MDS assembler from Apple and with some compilers (note: MDS is now shipping and includes a copy of *Inside Macintosh*). Get a copy of RMaker as soon as possible (there is a 10-page document you'll also need).

Recently, two editors of resources have become available. These are "Resource Editor" and "Redit". Both will edit resource data, often in a friendly format. For instance, the Resource Editor will allow you to edit an icon fatbits style and not as 128 bytes.

If you bought *Inside Macintosh* and the software supplement, these tools were sent to you. If not, there are still some places to get them. MacTutor Utility disk #1 has resource tools (see inside front cover for ordering).

### **The Dialog Manager**

Now we will get back to the problem at hand: how to make a box that asks for input. What we ultimately need is a window (which has a grafport) and some controls. We also will need to use the text editor to edit the input data. If it weren't for the Dialog Manager this would be real work. Fortunately, the Dialog Manager will perform all of these chores for us. I won't go too deeply into this because that is what IM is for, but all you have to do is give the dialog manager an id to a resource of type DLOG and, assuming that the data is in the correct form, everything is automatic.

### **Classy\_Input.R**

The first step in our project is to create a file with three resources in it. Type Classy\_Input.r (below) into a text file and apply RMaker to it. After that, use RMover to look and see if the resources exist in the file as planned. If you don't have RMaker you might try making the file with Resource Editor. If there is a lot of trouble creating this file (it's important), write to me care of MacTutor and I will do a

resource creation project. Actually, we should use RMaker input files as the official resource specification and exchange method.

### Classy\_Input.Pas

Next, boot MacPascal and type in Classy\_Input.pas (below). This is a very simple program. First we open our resource file. The dialog box is created by GetNewDialog, operated by ModalDialog, and erased by DisposDialog. Simple! Of course, I've gotten fancy and used GetDItem and SetIText to change the text of some items. It is also necessary to use a GetDItem-GetIText combination to get the text from the edittext item. With these same simple routines it is possible to do much more elaborate dialogs. As a matter of fact, you can completely change the appearance of this example by merely changing the resource specification. Note that this is the same method used by real Mac applications and is not just a kludge for educational purposes.

You may omit Prompt\_For\_Number at first if you wish. After you have Prompt\_for\_String working you may wish to try for numbers. The prompt for strings looks like this:

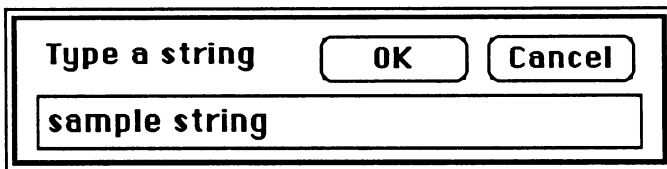


Fig. 1 Program Sample

To use this same box to prompt for numbers, we will use Package #7. To access a package you use its inline trap and leave a selector on the stack: eg. inlineP(TRAP,selector). However, Pack#7 is register based so it is necessary to use the register-based interface that I presented in MacTutor No.6.

GetNewDialog accesses resource DLOG 12345 and DITL 12345 in order to determine the characteristics of the box it will make. Pack#7 loads resource PACK 7 (in the file "System"), locks it on the heap and then calls it as a procedure. (We will call our own procedures the same way in later articles). By far the simplest of the resource-using toolbox routines is GetString:

```

procedure GetString(ID:integer):Handle;
begin
 GetString:=GetResource('STR ',ID);
end

```

This also one of the simplest traps of any kind. All text of any kind should be in resources for later translation. This is the reason we use GetString, and also the reason we use GetNewDialog and not NewDialog. You should develop the habit of using resources this way, too.

### Classy\_Input.R

```

;; Resource data for Classy_Input
;;
;;
;; Process this TEXT file through
;; RMaker:the Resource Compiler
;; by Andy Hertzfeld.

;; double semicolons start comments
;;
MacTutor_disk:Classy_Input.rsrc;;
????????
;; destination volume:filename
;; type and creator of file = ????

;; The first resource is the
;; definition of a Dialog Manager
;; window.

;; global coordinates !!

type DLOG
box,12345
;; note we have no title
96 128 148 384 ;; top left bottom right
visible goaway
1 ;; window type = dBoxProc
0 ;; refcon
12345 ;; ID of DITL associated
;; with this DLOG

;; Next is a list of 'items' to go
;; in the window.

type DITL ;; see Dialog Manager
items,12345
4 ;; four items

Button
4 120 24 180 ;; local coordinates !!
OK;;

Button
4 188 24 248
Cancel;;

EditText Disabled
32 8 48 248
*****text added later

StaticText Disabled
4 8 20 120
Type a String ;; prompt text, modified later

;; disabled means that the Dialog Manager
;; will not report events in this control
;;

;; And finally, the prompt used by
;;

```



```

.. ** Prompt_For_Number **
.. *****
..
type STR
prompt,12345
Type a number

.. *****
.. ** end of file **
.. *****

Classy_Input.pas
program Classy_Input_test;{ by Alan Wootton 5/85 }
type
 Ptr = ^integer;
 Handle = ^Ptr;
 StrPtr = ^str255;

var{ for test, not a part of final logic }
 rid : integer;
 number : longint;
 words : str255;

function Prompt_for_Str (prompt, sample : str255) :
 str255;
var
 DlogPtr : Ptr;
 itype, itemHit : integer;
 r : rect;
 itemH : Handle;
 tempStr : str255;

{ This uses DLOG ID=12345 and associated DITL with }
{ the same ID. Items are: #1=OK , #2=Cancel, }
{ #3=result, #4=prompt. First we will _GetDitem to }
{ obtain handle and then _SetIText. For item#3= result }
{ string, and item#4 = prompt string. This will leave }
{ itemH referring to result item. }
begin
 DlogPtr:= pointer(Lineref($A97C,12345,nil,pointer(-1)));
 { _GetNewDialog }

 inlineP($A98D, DlogPtr, 4, @itype, @itemH, @r);
 { _GetDitem }
 if prompt <> " then
 inlineP($A98F, itemH, @prompt);{ _SetIText }

 inlineP($A98D, DlogPtr, 3, @itype, @itemH, @r);
 { _GetDitem }
 if sample <> " then
 inlineP($A98F, itemH, @sample);{ _SetIText }

 inlineP($A991, nil, @itemHit);
 { _ModalDialog, returns with 1 or 2 in itemHit }

 if itemHit = 2 then { if Cancel }
 tempStr := "
 else { if OK }
 inlineP($A990, itemH, @tempStr);
 { _GetIText from item#3 }

 inlineP($A983, DlogPtr);{ _DisposDialog }

```

```

 Prompt_for_Str := tempStr;
end;

function Prompt_for_Number (sampN : longint) :
 longint;
var
 sHan : ^StrPtr;
 str : str255;
 access : array[0..12] of integer;
 d0, a0 : longint;{ the 68000 registers }
{ What we will do here is convert the input number }
{ into a string and then pass it to Prompt_for_String. }
{ Then we convert the returned string into a number. }
{ The conversions are done with $A9EE=pack7 . }
{ See Packages. }
{ Access is a 2 register OS interface that we must }
{ use because the call is register based yet also requires }
{ for the selector (the 1 or 0) to be on the stack }
{ (and 'Generic' won't do that) }
begin
 stuffHex(@access,
'2848548C41FA000C309F245F265F20522013FFFF224826804
ED4');
 a0 := ord(@str);
 d0 := sampN;
 inlineP($4E75, 0, @d0, @a0, $A9EE, @access);
{ pack7,0 is _NumtoString }
{ Str is now the string of sampN }

 sHan := Pointer(Lineref($A9BA, 12345));
 { _GetString, numeric prompt }
 str := Prompt_for_Str(sHan^^, str);
 { Str is now new string, convert d0 to num. }
 { A0 is still @str }

 inlineP($4E75, 1, @d0, @a0, $A9EE, @access);
{ pack7,1 is _StringtoNum }
 Prompt_for_number := d0;
end;

begin{ main procedure, used only to exercise }
{ procedures above. }

 rid := WinlineF($A997, 'MacTutor_disk:Classy_Input.rsrc');{
 _OpenResFile }
 { this file is the output of Rmaker, we open it to access }
 { our resources }
 if rid < 0 then
 writeln('error opening file', rid)
 else
 begin
 words := Prompt_for_Str('Type a string', 'sample string');
 writeln(' the words returned are:', words);

 words := Prompt_for_Str("", "");{ no prompts }
 writeln(' the words returned are:', words);

 number := Prompt_for_Number(1234);
 writeln('the number returned is', number);
 end;
 inlineP($A99A, rid);{ _CloseResFile }
end.

```



# Pascal Procedures

## All About Printing



Macintosh Pascal

Alan Wootton

MacTutor Contributing Editor

MacTutor Vol. 1 No. 9

This month's topic is printing. As an example I present a program that will print a message in large, 'banner' format. Perfect for parties!

To those of us who are familiar with the older generation of computers, printing is a matter of sending data through ports and using printer control codes. For the Mac we take these details for granted and will concentrate on using pre-packaged drivers and on the details of GrafPorts.

### Background

Printing on the Mac is done through the *Printing Manager*, RAM software that is considered part of the low-level operating system software. This code is stored in the system file and loaded into RAM. The Printing Manager interfaces between your program and the actual printer driver that controls the printer. The Imagewriter and LaserPrinter files are printer drivers called by the Printing Manager. The "Choose Printer" desk accessory switches between the two drivers so the Printing Manager has the correct driver installed and ready to go. To be more specific, the Imagewriter and LaserPrinter files are *printer resource files*, which contain device driver code within them to drive the respective printer. The "Choose Printer" DA copies the selected printer driver and its data structures from the printer resource file into the system file, and that's the driver that the Print Manager then uses. The idea is that a Mac application can be written using printing manager calls and be printer independent, since the printing manager would then interface to the appropriate device driver, eliminating the need to re-code an application for a different printer. This ideal comes close to being achieved on the Mac, probably more than any other computer system, but is still far from perfect.

The intent of the Mac designers was that printing would be transparent to quickdraw graphics "printed" on the screen. This was accomplished by using a special quickdraw grafport for printing. Normal quickdraw routines are used to draw into this port. The Printing Manager converts these quickdraw calls in your grafport to printer calls to drive the printer instead of the screen. In this way, printing is supposed to be transparent to a normal screen drawing operation. It also explains the complexity of printing, since it involves all the quickdraw screen graphics technology.

### Draft Versus Standard Quality

The Printing Manager is responsible for the dialog boxes

that appear prior to printing and for allowing draft or standard quality printing. These two modes are fundamentally different. In draft mode, the ascii representation of the quickdraw calls are sent to the printer, which prints the text of the document using the character generator within the printer. Hence no graphics are supported except character graphics or modes directly supported by the printer and driven by the user's program. The printer responds to its own command codes rather than any of the "Mac" like graphics.

### Spool File Versus Printing the Spool File

Standard and high quality modes are completely different. In these modes, the printer is controlled bit by bit and the entire document is converted into a giant bit mapped graphics dump. A temporary quickdraw picture is first composed on disk called the *spool file*. This spool file is then read into memory, converted into an array of dots by *bands*, and sent to the printer. The bands are necessary because of the huge amount of memory required by bit-mapping an 8 1/2 by 11 inch page. For the imagewriter, this amounts to 256K of memory. For the LaserWriter, over a megabyte! The task of first creating a quickdraw picture for the spool file, and then *imaging* (converting to a bit-map) and printing the spool file band by band, are handled separately by the printing manager. In fact, it is supposed to be possible to create several spool files for printing later, but no one seems to be using that ability. Inside Mac appears indecisive about printing, saying that printing spool files is done by the *Printer* utility, the code of which has apparently been integrated into the Finder. But Printer does exist as a separate utility that will read a spool file, image it and print it. Since hardly any applications leave the spool file lying around, there hasn't been much opportunity to use the Printer utility.

### Print Manager Calls: Lisa Pascal to Mac Pascal

In the printing chapter of *Inside Mac*, a lengthy list of data types is presented along with 14 Pascal procedures and functions. This is the printing interface, but it is not made clear exactly where these procedures are found. The idea is that you are supposed to simply include the declarations and calls in your program and then link with a printing module. What is misleading about this is that the link is not with the procedures you think you are using but merely a piece of "glue". The actual code used to drive the printer is in the file 'Imagewriter' (usually in the system folder). Naturally, the code is in resources.

## Imagewriter

To get a handle (pun intended) on printing, I examined the link module provided by Owen Densmore. I then duplicated Owen's algorithms in Pascal. You can find my versions of all 14 procedures in the following program.

The first task is to open a resource file. I said a bit on resources last month and there is never enough I can say about them. You should take a look in 'Imagewriter' and see what is there. The resources of interest here are of TYPE PDEF. These resources are actual printer-specific code. The code we need is in PDEF#0, PDEF#1, PDEF#4, and PDEF#5. In 'Imagewriter' is also a driver DRV#-8192 named, 'print', but the one actually used is installed in the file, 'system'. The other stuff is for support of the dialog boxes the print manager presents. Rather than simply *OpenResFile('Imagewriter')*, we must take into account the possibility that there is more than one printer. To this end there is a resource STR #-8192 which contains the name of the print resource file. Therefore the print procedure *PrOpen* is *ResID := OpenResFile( GetString(-8192) );* along with *PBOpen* of 'Print'. *PrClose* is simply *CloseResFile(ResID)*. The other print procedures are in PDEF resources and begin at various offsets into those resources.

The function *GetAddress* is used in the program to load one of the PDEF's (*GetResource*) and then to lock it in place (*Hlock*). It is then dereferenced and the offset is added. Once we have an absolute address to use we need a way to pass control to it. To do this I created the array *jsr[0..3]* of integer which contains code to JSR to the address on the top of the stack. Once again I take advantage of the fact that *InlineP* leaves register A0 pointing 2 bytes behind of the place we must return to. If you know no 68000 assembly, you can ignore this.

The final detail to emulating the print interface code is that some of the routines release the PDEF handle and others leave it locked in place. This is the purpose of *ReleaseAddress*. Further, note that some of the routines use different PDEF's (0,1,2 or 3) depending on the setting of *bjDocLoop*. This is the purpose of *MethodAdd*. I find this a rather obscure feature and it will not be used here; nonetheless, the interface presented is complete, I think.

## GrafPorts for Printing

It is necessary to use the type declarations to print but it is not necessary to closely study their form or function. I suggest studying my print record declarations in the program below along side "Printing Resources" in the new telephone version of *Inside macintosh*, where the print data structures are explained. A print record is created and filled in by the Printing Manager and the print dialog boxes. The printing port is set by a variant record type so that your program can write into the port as if it were a regular grafport, while the Printing Manager can modify the grafport for the printer.

The central issue is one of drawing in a GrafPort. Essentially, printing is no different than updating the contents of a window except that the size of the window is huge. To simply print a picture use a sequence like *PrOpen*, *PrintDefault*, *PrOpenDoc*, *PrOpenPage*, draw your picture, *PrClosePage*, *PrCloseDoc*, *PrPicFile*, and *PrClose*. To print more pages repeat *PrOpenPage*, draw, *PrClosePage*. Note the perfect symmetry with the exception of *PrintDefault* and *PrPicFile*. *PrintDefault* will fill in all those nasty hPrint fields for you and *PrPicFile* actually drives the printer. You may also modify your hPrint with *PrStdDialog* and *PrJobDialog*. These two present very familiar dialog boxes.

In the following program the procedure *DrawPic* draws a large picture in whatever port is current. In the main body of the program *DrawPic* is called several times to demonstrate the use of *SetOrigin* to show different views of the big picture. Of course, the parts of the picture that don't fit are chopped off by *quickdraw* and the same will happen when we print. I have found some strange complications. When printing, the *VisRgn* of the print port is not used, rather the *clipRgn* is used to delineate the usable area. In contrast, the drawing example opens up the *ClipRgn* and the *VisRgn* presents the limits. To investigate put:

```
with MyPort^.VisRgn^.rgnBBox do
writeln(top,left,bottom,right);
```

in various places.

If you are doing a picture that is larger than one page, don't assume the size of the page. You won't have to mess around in the grafport because you can use the rectangle *hPrint^.prInfoPt.Rpage* to obtain the necessary offsets. *Rpage.top* and *Rpage.left* are zero so use right and bottom. I think this is the only one of the many hPrint fields that is commonly useful.

To make a banner, as promised, you will need the newer version of Imagewriter (sometimes called Imagewriter 15) or else there will be breaks between the pages. For banner format choose 'wide' and 'no breaks between pages' on the style dialog. You can change *DrawPic* to make any picture you please.

If there is any, and I mean any, subject that you think needs to be covered, write to me care of MacTutor and I will give it a try. I figure that if we can print with MacPascal then MacPascal can do just about anything.

```
program Banner_Print;{ by Alan Wootton 5/85 }
{ Prints a test pattern using the Printing Manager }
```

```
uses
 Quickdraw2;
type
 ptr = ^integer;
 handle = ^ptr;
 ProcPtr = ^longint;
 OStyle = longint;
```

```

strptr = ^str255;
strHan = ^strptr;

{~~~~~ Print Manager Data Types ~~~~~}
{ note: byte size fields don't work right in MacPascal }

TPStr80 = ^TStr80;
TStr80 = string[80];

TPRect = ^Rect;

TPPrPort = ^TPPrPort;
TPPrPort = record
 gPort : GrafPort ;{ GrafPort to draw in }
 gProcs : QDProcs ;{ pointers to drawing routines }
 LGParam1, LGParam2 : longint;
 LGParam3, LGParam4 : longint; { internal use }
 { fOurPtr:boolean }
 fOurBits : integer; { boolean }
end;

TPPort = record
 case integer of
 0 : (
 pGPort : GrafPtr
);
 1 : (
 pPrPort : TPPrPort
)
 end;

TPPrInfo = record
 iDev : INTEGER; {driver information}
 iVRes : INTEGER; {printer vertical resolution}
 iHRes : INTEGER; {printer horizontal resolution}
 rPage : Rect {page rectangle}
end;

TPPrStl = record
 wDev : integer; { TWord; used internally}
 iPageV : INTEGER; {paper height}
 iPageH : INTEGER; {paper width}
 { bPort : SignedByte; printer or modem port}
 TFeed : integer; { TFeed; paper type}
end;

TFeed = (feedCut, {hand-fed, individually cut sheets}
feedFanfold, {continuous-feed fanfold paper}
feedMechCut, {mechanically fed cut sheets}
feedOther); {other types of paper}

TPPrJob = record
 iFstPage : INTEGER; {first page to print}
 iLstPage : INTEGER; {last page to print}
 iCopies : INTEGER; {number of copies}
 bJDocLoop : integer;
 { printing method (in upper byte) }
 { fFromUsr : BOOLEAN; }
 { TRUE if called from application }
 pIdleProc : ProcPtr; {background procedure}
 pFileName : TPStr80; {spool file name}

```

```

iFileVol : INTEGER; {volume reference number}
{
 bFileVers : SignedByte; }
{
 version number of spool file }
bJobX : integer; { SignedByte not used}
end;

```

```

TScan = (scanTB, {scan top to bottom}
scanBT, {scan bottom to top}
scanLR, {scan left to right}
scanRL); {scan right to left}

```

```

TPrXInfo = record
 iRowBytes : INTEGER; {bytes per row}
 iBandV : INTEGER; {vertical dots}
 iBandH : INTEGER; {horizontal dots}
 iDevBytes : INTEGER; {size of bit image}
 iBands : INTEGER; {bands per page}
 { bPatScale : SignedByte; used by Quickdraw}
 bUIThick : integer;
 { was Signed Byte, underline thickness}
 { bUIOffset : Signed Byte; underline offset}
 bUIShadow : integer;
 { was SignedByte, underline descender}
 scan : integer; { TScan(byte), scan direction }
 { bXInfoX : SignedByte not used }
end;

```

```

THPrint = ^TPPrint;

```

```

TPPrint = ^TPrint;

```

```

TPrint = record
 iPrVersion : INTEGER; {Printing Manager version}
 prInfo : TPPrInfo; {printer information}
 rPaper : Rect; {paper rectangle}
 prStl : TPPrStl; {style information}
 prInfoPT : TPPrInfo; {Pcopy of PrInfo}
 prXInfo : TPrXInfo; {band information}
 prJob : TPPrJob; {job information}
 printX : array[1..19] of INTEGER
 { printX used internally by print manager}
end;

```

```

TPrStatus = record
 iTotPages : INTEGER; {total number of pages}
 iCurPage : INTEGER; {page being printed}
 iTotCopies : INTEGER; {number of copies}
 iCurCopy : INTEGER; {copy being printed}
 iTotBands : INTEGER; {bands per page}
 iCutBand : INTEGER; {band being printed}
 { fPgDirty : BOOLEAN; in lower byte of iCutBand }
 { dirty is TRUE if started printing page}
 fImaging : integer; { BOOLEAN; TRUE if imaging}
 hPrint : THPrint; {print record}
 pPrPort : TPPrPort; {printing port}
 hPic : PicHandle {used internally}
end;

```

```

{ Parameter Block information contained in File Manager}
{ chapter of Inside Macintosh. Note that MacPascal }
{ won't do 8 bit fields right. We will only use the }
{ ioParam part here }

```

```

ParamBlkPtr = ^ParamBlockRec;
ParamBlockRec = record
 qLink : Ptr;
 qType : integer;
 ioTrap : integer;
 ioCmdAddr : ptr;
 ioCompletion : ptr;
 ioResult : integer;
 ioNamePtr : ^str255;
 ioVrefNum : integer;
{ case ParamBlkType of ... ioParam: }
 ioRefNum : integer;
 { ioVersNum : byte; }
 ioPermssn : integer;{ byte }
 ioMisc : ptr;
 ioBuffer : ptr;
 ioReqCount : longint;
 ioActCount : longint;
 ioPosMode : integer;
 ioPosOffset : longint;
end;

var {-----global variables-----}
{ jsr and access are 68000 glue routines }
{ jsr source code described below }
{ See MacTutor vol.1 no.6 for access source code }
jsr : array[0..3] of integer;
access : array[0..12] of integer;
ResId : integer;{ id of currently open Printer file }
hPrint : THPrint;{ data record for print job }
pPrPort : TPrPort;{ pointer to port to draw into }
PrStatus : TPrStatus;{ printing status record }
page : integer;{ page number currently printing }
pageR : rect;{ copy of rPaper rectangle }
width : integer;{ width of paper }
maxR : rect;{ huge rectangle }
myport : GrafPtr;{ for temporary use }

{ %%% %%% %%% %%% }
{ % % % % % % % % }
{ %%% %%% % % % % }
{ % % % % % % % % }
{ % % % % % % % % }
{-----beginning of procedure definitions-----}

{ This loads, locks, dereferences, and computes an }
{ entry point for a resource that will be used as code }
function GetAddress (id, offset : integer) : longint;
var
 h, d : Handle;
begin
 h := pointer(LinlineF($A9A0, $50444546, id));
 { _GetResource('PDEF',id) }
 d := nil;
 inlineP($4E75, @d, @H, $A029, @access);
 { _Hlock(H) }
 if d <> nil then
 writeln('GetAddress error', ord(d));
 GetAddress := ord(h^)+ offset;
end;

```

```

{ This unlocks a PDEF resource that was locked to run }
procedure ReleaseAddress (id : integer);
var
 h, d : Handle;
begin
 h := pointer(LinlineF($A9A0, $50444546, id));
 { _GetResource('PDEF',id) }
 d := nil;
 inlineP($4E75, @d, @H, $A02A, @access);
 { _Hunlock(H) }
 if d <> nil then
 writeln('ReleaseAddress error', ord(d));
end;

{ this calls GetAddress for the appropriate resource. }
{ Which resource is used depends upon bjDocLoop. }
function MethodAdd (hPrint : THPrint;
 offset : integer) : longint;
var
 method : integer;
begin
 method := hPrint^.prJob.bjDocLoop div 256;
 method := method mod 4;
 hPrint^.prJob.bjDocLoop := method * 256;
 { bjDocLoop is in upper byte. It must be mod 4 }
 MethodAdd := GetAddress(method, offset);
end;

{common OS trap code, could be done with 'Generic' call}
{ see MacTutor vol.1 no.6 for source code }
function filecall (Pb : ParamBlkPtr;
 trap : integer) : integer;
var
 d0, a0 : longint;
begin
 a0 := ord(pb);
 inlineP($4E75, @d0, @a0, trap, @access);
 { $4E75 is rts to access routine }
 filecall := loword(d0);
end;

{ The following File Manager calls work just like }
{ those described in Inside Macintosh for the Lisa Pascal }
{ Workshop, except that the async parameter is a }
{ dummy; all calls are sync. }

function PBOpen (Pb : ParamBlkPtr;
 async : boolean) : integer;
begin
 PBOpen := filecall(pb, $A000);
end;

{ Below are the Printing Manager calls. }
{ They are supposed to work just like the real ones. }

procedure PrOpen;
var
 sH : strHan;
 pblock : ParamBlockRec;
 Tstr : str255;
begin
 Tstr := '.Print';

```

```

pBlock.ioNamePtr := @Tstr; { first we open driver...}
pBlock.ioPermsn := 0;
If PBOpen(@pBlock, false) <> 0 then
 sysbeep(100);
{...then we open the resource fork of current printer file}
sH := pointer(LinlineF($A9BA, $E000));
{ _GetString(-8192) = name of print manager }
ResId := WinlineF($A997, sH^);
{ _OpenResFile of print manager, usually='Imagewriter'}
end;

```

```

procedure PrClose;
begin
 inlineP($A99A, ResId); { _CloseResFile }
 { note that the driver is left open }
end;

```

```

procedure PrintDefault (hPrint : THPrint);
begin
 inlineP($4E75, hPrint, Getaddress(4, 0), @jsr);
 { $4E75 is rts to 'jsr' routine which runs }
 { PDEF #4 with offset of zero. }
 { This same sequence is used below also }
 ReleaseAddress(4);
end;

```

```

function PrValidate (hPrint : THPrint) : boolean;
begin
 PrValidate :=
 BinlineF($4E75, hPrint, Getaddress(4, 24), @jsr);
 ReleaseAddress(4);
end;

```

```

function PrStlDialog (hPrint : THPrint) : boolean;
begin
 PrStlDialog :=
 BinlineF($4E75, hPrint, Getaddress(4, 4), @jsr);
 ReleaseAddress(4);
end;

```

```

function PrJobDialog (hPrint : THPrint) : boolean;
begin
 PrJobDialog :=
 BinlineF($4E75, hPrint, Getaddress(4, 8), @jsr);
 ReleaseAddress(4);
end;

```

```

procedure PrJobmerge (hPrintSrc, hPrintDst :
 THPrint);
begin
 inlineP($4E75, hPrintSrc, hPrintDst,
 Getaddress(4, 28), @jsr);
 ReleaseAddress(4);
end;

```

```

function PrOpenDoc (hPrint : THPrint;
 pPrPort : TPPrPort;
 pIOBuf : Ptr) : TPPrPort;
var
 Ill : longint;

```

```

begin
 Ill := LinlineF($4E75, hPrint, pPrPort, pIOBuf,
 MethodAdd(hPrint, 0), @jsr);
 PROpenDoc := pointer(Ill);
end;

```

```

procedure PrOpenPage (pPrPort : TPPrPort;
 pPageFrame : TPRect);
begin
 inlineP($4E75, pPrPort, pPageFrame,
 MethodAdd(hPrint, 8), @jsr);
end;

```

```

procedure PrClosePage (pPrPort : TPPrPort);
begin
 inlineP($4E75, pPrPort, MethodAdd(hPrint, 12), @jsr);
end;

```

```

procedure PrCloseDoc (pPrPort : TPPrPort);
begin
 inlineP($4E75, pPrPort, MethodAdd(hPrint, 4), @jsr);
 ReleaseAddress(1);
end;

```

```

procedure PrPicFile (hPrint : THPrint;
 pPrPort : TPPrPort;
 pIOBuf : Ptr;
 pDevBuf : Ptr;
 var prStatus : TPrStatus);
begin
 inlineP($4E75, hPrint, pPrPort, pIOBuf, pDevBuf,
 @prStatus, Getaddress(5, 0), @jsr);
 ReleaseAddress(5);
end;

```

```

function PrError : integer;
var
 eptr : Ptr;
begin
 eptr := pointer($944);
 PrError := eptr^;
end;

```

```

procedure PrSetError (iErr : integer);
var
 eptr : Ptr;
begin
 eptr := pointer($944);
 eptr^ := iErr;
end;

```

```

{ This is my routine to draw a sample picture. }
{ The picture is drawn into the current port. }
{ Note that this is a very wide picture. }
procedure drawpic;
var
 i : integer;
 r : Rect;
begin
 moveto(4, 220);

```

```

textfont(2);
textsize(216);{ must be less than 256 }
textface([bold, outline, underline]);
drawstring('Hooray for MacTutor!');
end;

 %%%%%%%%%%
{ %%%
{ %
{ Main entry point of program
begin
stuffHex(@jsr, '5488225F2F084ED1');
{ Code to jsr to address on top of stack: }

{ 5488 addq.l #2,a0 ;a0 is now return address }
{ 225F move.l (a7)+,a1 ;a1 is address of routine }
{ 2F08 move.l a0,-(sp) ;put return on stack }
{ 4ED1 jmp (a1) ;and call routine }

stuffHex(@access,
'2848548C41FA000C309F245F265F20522013FFFF224826804
ED4');
{ access calls a register based trap }

showdrawing;{ Set current port to Drawing window. }
setrect(maxR, 10, 100, 510, 330);
SetDrawingRect(maxR);{ Make Drawing big. }
setrect(maxR, -32000, -32000, 32000, 32000);
{ make maxR huge }

cliprect(maxR); { Open up clip region. }
for page := 0 to 8 do{ Draw 9 times for preview. }
begin
setOrigin(page * 300, 0);{ Use increasing offset. }
eraserect(maxR);
drawpic;{<----- Draw our picture in drawing. }
end;

hPrint := NewHandle(120);
{ 120 is size of TPrint record. }

PrOpen;{ Open Printing Manager. }

PrintDefault(hPrint);{ Fetch default hPrint. }

{ Now modify style info of hPrint if desired. }

{ For banner chose Wide and No Breaks Between Pages. }
If PrStdDialog(hPrint) then
writeIn('new style chosen');

{ Then get job info for hPrint. }
If PrJobDialog(hPrint) then
begin
pPrPort := PrOpenDoc(hPrint, nil, nil);
{ Grafport is now pPrPort, not Drawing window. }

pageR := hPrint^.prInfoPt.Rpage;
pageR.right := pageR.right div 2;
pageR.bottom := pageR.bottom div 2;
width := pageR.right;{ width of printing 'window' }
{ Draw into rect 1/2 page size, quickdraw will }
{ scale picture to fit page later. This effectively doubles }
{ the size of our drawing. }

{ Now we will draw the picture. It will go into pPrPort. }
{ For each page we offset to clip off parts not used. }
getport(myport);{ myport := pPrPort; }
for page := 0 to 7 do{ 8 pages }
begin
PrOpenPage(pPrPort, @pageR);
{ PrOpenPage resets the ports size, coordinates etc. }

setOrigin(width * page, 0);

{ setOrigin does not change cliprgn so we do it here. }
offsetRgn(myport^.cliprgn, width * page, 0);
{ I tried simply maxing out the clip and it didn't work. }

drawpic;{<-----draw picture into pPrPort}

PrClosePage(pPrPort);
end;

PrCloseDoc(pPrPort);

PrPicFile(hPrint, nil, nil, nil, PrStatus);
end;{ if job }

disposeHandle(hPrint);
PrClose;
end.

```



# Pascal Procedures

## Dial a Fortune Apple Talk Client



Alan Wootton  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 10

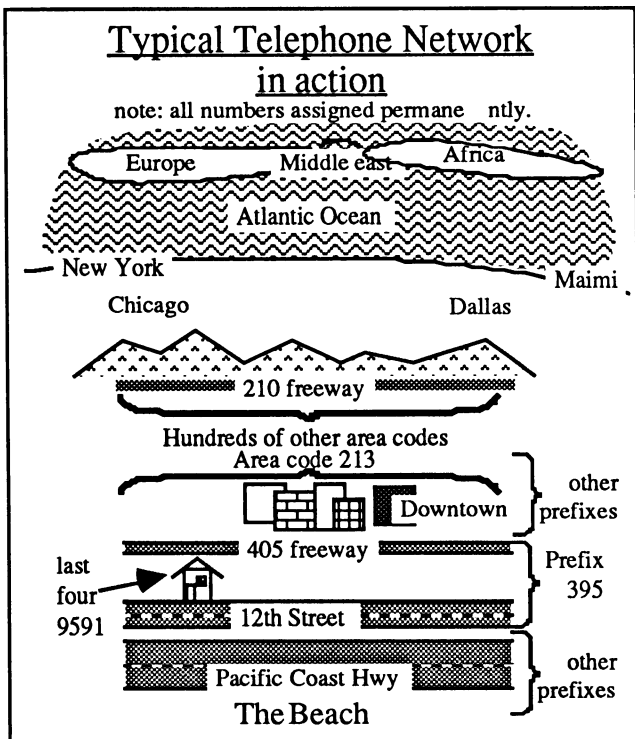


Fig. 1 Phone-type Network

### AppleTalk and Macintosh Pascal

This month's topic is AppleTalk. In collaboration with Bob Denny (see the column 'C Workshop' in this issue), I present a pair of programs that form 'Dial-A-Fortune'. The names of the two programs presented below are `Get_Fortune` and `Send_Fortune`. These are fully conversational with Bob Denny's server version, which performs the task of `Send_Fortune`. Of course, to try them, you will need two Macs. Either my Pascal `Send_Fortune` or Bob's server program run in one Mac while my Pascal `Get_Fortune` runs in the other Mac as the "requestor". `Get_Fortune` will request fortunes from either of the C or Pascal `Send_Fortune` servers which respond by sending back a fortune cookie! In this way, all the concepts of AppleTalk are illustrated. (Note: due to space limitations, the C version of Bob's requestor program was not included in the C column. Use my `Get_Fortune` requestor instead. A C requestor program to compliment Bob's server will be published in the C workshop next month.) If you are using Pascal, everything you need is here in this column for two way AppleTalk communications.

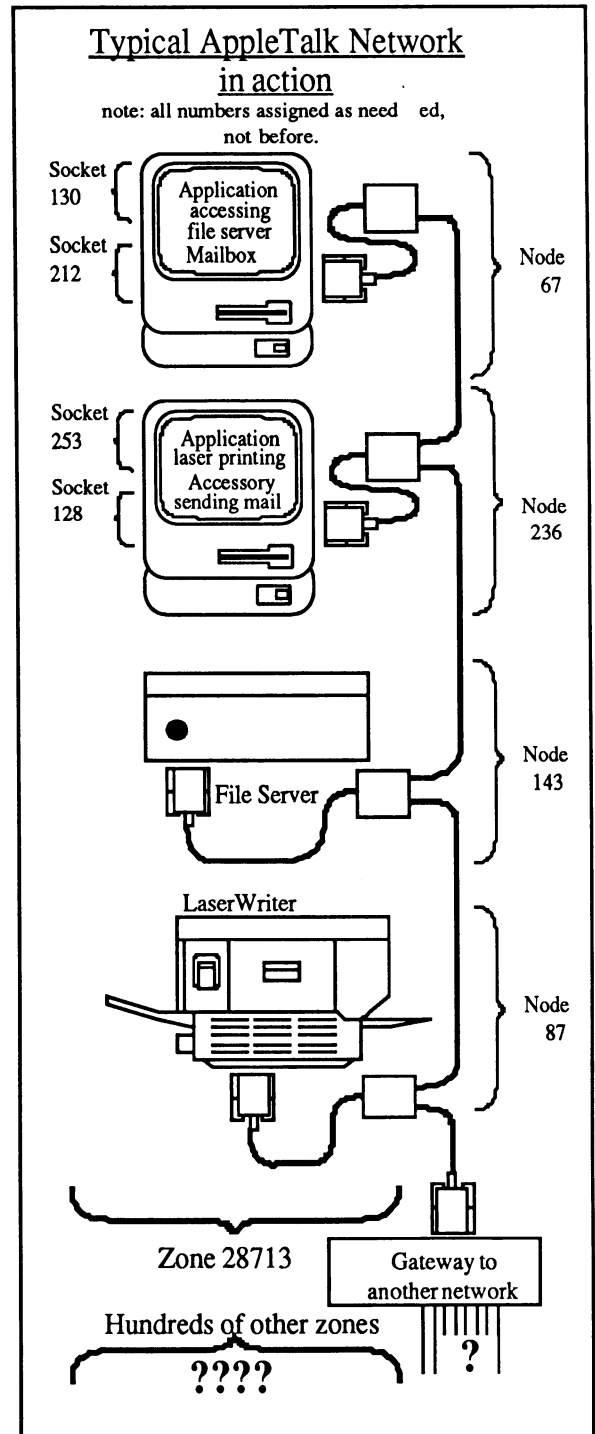


Fig. 2. AppleTalk-type Network



## Appletalk

I started this project with only a passing knowledge of networks. Networks are for hooking computers together, right? So, I went to the store and bought one -- actually, two -- one for each computer. After paying \$50 retail for each, I hurried home to try it out. Shock! What you get are merely cables to wire your machines together, and a short pamphlet on how this is done. A child could do it, but having the wires in place does not a network make. You need software.

I referred immediately to Inside Macintosh. After careful study I identified two types of software that are needed. There are some drivers that you install in your system like a desk accessory, or like a new printer driver, and there are Macintosh Programs that use the drivers. Searching through the billions of disks sent to the Apple Certified Developers uncovered the drivers (presumably you can get them at the store too), and I installed them. Still, no network. Actually, it was like having telephones but not knowing how to talk.

I could have gone to the store to look for software to use my network, but I decided to write my own (actually, Bob Denny gave me Dial-A-Fortune first). So, back to the Bible -- excuse me -- Inside Macintosh.

What I found are many programs sending many messages at many levels of organization. The drivers send and receive data over the net at the drop of a hat. On the top of them all seemed to be a thing called AppleTalk Transaction Protocol, ATP for short. What this is, are some routines you can call to send and receive data (up to 512 bytes at a time). ATP is pretty clever. It will wait until it is its turn to talk, it will retransmit requests if no reply is received, and more. In the descriptions of the routines, I found numbers to set to indicate what data to send, and how persistent I wanted ATP to be about delivery. I found something else, too.

To use a network to 'hook computers together' you have to know something about the rules and conventions. Just as you cannot mail a letter without understanding about countries, cities, streets, addresses, and names, you also cannot make a phone call without knowing when you need an area code, or when you need a prefix, or when you can just dial 8 and then an extension. As I found, you must tell ATP three things to have a message delivered (refer now to the diagrams of networks in action).

First is the network number or zone (an integer). This is for situations where your network is connected to other networks. For me this is like an area code in a world with two telephones -- not real useful. After ATP knows which network you're calling, it wants to know which node you want (a byte). I had two nodes. Each computer with a serial port, running the drivers, is a node. This is as if each of my computers were a whole town, each meriting its own prefix, even though there may be only one phone in each town.

Simple so far, but now my misconceptions begin to show. This network is for connecting *programs* together, not computers. The final destination of any message is a socket. For this ATP requires a socket number (a byte. Personally, I think that 'socket' is a poor choice of words for this; it brings the image of a connector to mind, not a software process). This is analagous to the last four digits of a phone number. Even though there is only one 'phone' in each of two 'towns' you still must 'dial' an 'area code' (net number), a 'prefix' (node number), and the 'last four' (socket number) for a 'call' to go through. So let's send some messages!

What is the number for the place we wish to call? Telephones are considered mostly permanent installations and are given permanent numbers (a very IBM type of mentality). Area codes especially won't change (even though this just happened to L.A. when they split the city into two areas where there was previously one). For AppleTalk the designers decided that numbers might change a lot and this would become a drag (it would be). So they decided that they would make it so that numbers could change all the time. This is a lot like a system where every time you plug your phone in the wall you call the phone company with your name and they give you a brand new number. Makes it pretty hard for some one to call you! But wait -- as the lazy ones among us have already guessed, all you have to do is call directory assistance before every call (a very Apple type of mentality). This way it doesn't matter what your friend's number is this week; you can always call him. AppleTalk works exactly this way, except directory assistance is called Name Binding Protocol, NBP for short.

When you call directory assistance the first thing they want to know is "Which city, please". NBP will search *all* the cities, or nodes, for the name you want, but you must provide a string for the zone (area code) you want. If you are calling directory assistance in another state you probably called first to get the area code and then called (area code) 555-1212 for the number. NBP will do this all in one inquiry. So, before every call you ask NBP for the number of the entity with a particular name, in a particular zone. NBP, unlike the phone company, will also look in the yellow pages. You see, there is another string you pass to NBP which is meant to be an identifier for the 'type' of party you wish to reach. This is really a superior system (are you listening Ma Bell?). All parties on the system are identified by their zone, their name, and their type. You can ask NBP for all the numbers in any combination of these. Think how nice it would be to dial national directory assistance and ask for all the Smiths in L.A. that are in the computer business. Only the names are used. The numbers are only a necessary evil to be ignored as much as possible.

Now that we know how to deal with the authorities involved, let's get down to the nitty gritty and see if we can establish communication. Everyone get out your copies of Inside Mac, boot your Macs, and follow along.

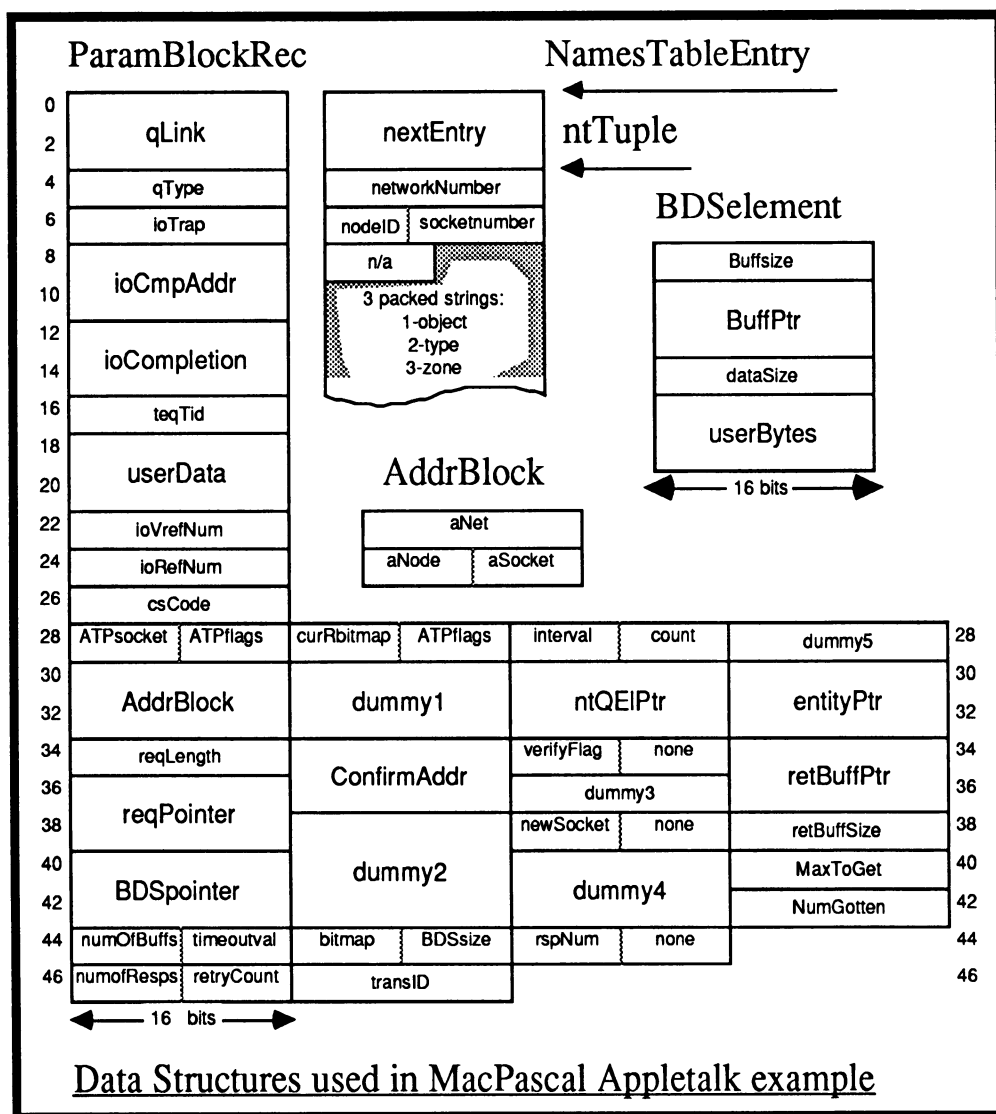


Fig. 3 Appletalk Data Structures

### Inside Inside Appletalk

It will be necessary to refer to the AppleTalk programmers guide. This is the 'Programmer's Guide' section in Inside Appletalk and the 'AppleTalk' section of Inside MacIntosh (regular or phone book version). They are all the same (first draft 1/31/85). Inside AppleTalk contains chapters describing in detail the many philosophies and protocols involved. The only routines needed are the Name-Binding protocol and the AppleTalk Transaction protocol.

The Appletalk section begins with brief descriptions of the various levels of service involved. You can ignore references to link access protocol (LAP) and datagram delivery protocol (DDP). On page 16 a description of how to call AppleTalk from Pascal begins. This is all for the Lisa-Pascal Workshop and is therefore completely worthless. Skim to page 66. Also ignore the AbusRecord type in the summary.

The data types we will use are all in the programs that follow and I present a diagram for easy reference. Don't worry about assembly language. We will simply use the parameter block lists presented after each routine name. The method of access to AppleTalk will be to make device manager control calls directly to the drivers involved. This is simply illustrated in the code that follows. Following NPB are instructions on page 78 for making your own handlers. We definitely don't need this so skip to the summary on page 86. Again, watch out for the 'Pascal' interface; we are using the assembly language information.

### ATP

The AppleTalk Transaction Protocol (ATP) is the main purpose behind the other, lower level, routines. ATP is used to send and receive messages on the system.

Although messages can be sent anonymously, every receiver has an address. The address (sometimes called the internet address) consists of a network number, a node number and a socket number. We will worry how to find an address later, so for

now assume this is taken care of. The network number is to specify which of several individual appletalk networks the receiver is on. This is presently not used. The node number specifies which Mac (or other processor) the message is to go to. The socket number specifies which of several entities (desk accessories for instance) the message is to go to.

To send a message somewhere on the network use the function `SendRequest`. Note that not only is a message sent but a reply is secured also. To use `SendRequest` you must know the address of the receiver. In our example the message sent is ignored and the reply contains a Pascal-style string which is a fortune.

In order to receive messages (not merely replies) you (meaning your program) must declare yourself as an address capable of receiving requests. To do this call `OpenATPSkt`. When you are done call `CloseATPSkt`. Once you have an open socket you may then ask to receive messages. To do

this call `GetRequest`. You may use asynchronous device manager calls in order to not tie up the system but in this example we will simply wait (hang) for a message. When a request is received call `SendResponse` to send the reply.

### Name-Blinding Protocol

Whenever a program opens a socket, ATP assigns an address to it. The address could be *any* valid address so it would be very hard for anyone to already know your address. To receive messages it is necessary to supply a *name* for that socket. Then another entity can call NBP with your name and NBP will find your address.

To do this call `RegisterName`. Each name is made of three strings. The strings are called the object string, the type string, and the zone string. The zone string is used to name which network one is on, and since multiple networks are not in common use '\*' is usually used for the zone string. The type string is to declare the purpose of this particular socket. Sockets could be used for receiving mail or for providing file access. Each function should use a unique type string. Our example uses 'Dial-A-Fortune' as the type. The name string is to differentiate between several sockets that all offer the same service. For example, my Dial-A-Fortune socket has an object name of 'MacPascal Fortunes', and Bob Denny's Dial-A-Fortune socket has the object name of 'Confuscious'. It might be wise to add a unique identifier, like the disk name, or the users name, on the end of this string (for instance 'Confuscious-Bobs disk').

Once you have called `RegisterName` it is much more likely that someone will send a message to your `GetRequest`. After you don't want anymore messages you should call `RemoveName` and then you may close the socket.

If you want to send a message and you don't know the address to use you can call `LookUpName`. You supply a name and `LookUpName` will find the address. As a matter of fact, you can look up many names. `LookUpName` will take wild card characters in the object, type, and zone fields. Pass an array to hold the resulting addresses and pass the maximum number of addresses that you want and `LookUpName` will try to fill it for you. Unfortunately, the 'array' of addresses also contains the names that go with them and the whole affair is packed together so that it is harder to access the individual elements in Pascal. Not that it couldn't be done, but for our example we only look up one name to avoid this problem.

### AppleTalk from MacPascal

As usual, the first task is to declare the types we will need. Our method of passing data to the various routines is to use a parameter block. This is the same block used by the file manager and the device manager. However, AppleTalk uses the fields in many new ways, and the various routines will use a particular field for up to four different purposes. To

accommodate this we define a parameter block with variant fields. A bad problem remains. MacPascal refuses to correctly lay out byte-sized fields. Observing that MacPascal will do packed arrays of char in the correct manner, and that byte-sized fields always occur in pairs, we will declare them in pairs and use an array index to choose the one we want. For example, to access `ATPsocket`, use `ATPsocket_ATPflags[0]`, and to access `ATPflags`, use `ATPsocket_ATPflags[1]`. A record of many variant fields is sometimes hard to refer to, so I have provided a diagram of all the records we will use and the positions of the fields therein in fig. 2.

The `NamesTableEntry` is another problem. The lower part of this record is three strings that have been packed together. The procedure `Pack3str` is used to accomplish this.

As I mentioned before, our access to the AppleTalk routines will be through device manager control calls. The routines presented set the `csCode` correctly, then call the correct driver interface. The driver interface then sets `ioRefNum` correctly and requests filecall to make a control call. Filecall was presented in previous columns in this magazine. Look in the past issues, or better yet, just use it the way I do and forget how it works.

Before you can try to get the examples running you must install AppleTalk into your system file. Apple supplies an installer that will do this (but it won't work on my HyperDrive!). The Appletalk installer moves these resources into the file 'System': `DRVR 9 .MPP`, `DRVR 10 .ATP`, `INIT 6`, and two of type `NBPC`. The `INIT` opens the `MPP` driver. The `NBPC` types are code used for NBP.

### Get\_Fortune

To `Get_Fortune` we first must check that the drivers are opened. This is the function of `ATPLoad`. I adapted this from the example on page 54. Its purpose is to open both drivers if they are not already open and if the serial port is free. I wish this were not necessary. It would be better if it were just a trap in ROM that you call which returns an OS error code. Obscure system globals are checked and `PBopen` calls are made. Let's just assume that it works and let the hot-shots analyze its code for extra credit.

Next, we stuff `Nblock` in preparation for a `LookUpName` call. Interval and count are the time between retries and the number of times to try, respectively. `Pack3str` is used to set the strings in `myNtable` and `entityPtr` is set to point at the string part of `myNtable` (not at the beginning). `RetBuffPtr` and `retBuffsize` specify where we want a `ntuple` returned, and `maxToGet` asks for only one to be returned. If you are following along in the book on page 76 you will note that I set the parameters in the order they are listed. Then `LookUpName` is called to return the first Dial-A-Fortune type it finds.

Now, myTuple will have an address in its upper part. After we set ATPflags for once-only-mode we transfer the three part address into Sblock. There will be no message so we set reqLength and reqPointer to nothing. Moving along, we set BDSpointer to our numOfBufs:=1 buffers. Timeoutval is the number of seconds that SendRequest will wait for a response before trying again, and retryCount is the number of times it will try before giving up. The parameter block is now filled but we first must set myBDS.buffPtr and buffSize to the address of the string reply which is where we want the fortune to be returned. After the SendRequest we examine reply and see that it does indeed contain a fortune transmitted from a Dial-A-Fortune somewhere on the network.

### Send\_Fortune

Send\_Fortune follows Get\_Fortune below but, since the upper portion of code is *exactly* the same as in Get\_Fortune, it is not repeated. *Please make note of this!* The addrBlock of Sblock, when cleared, indicates to OpenATPSkt that we want it to assign a new address to us (I can't think of a situation where we would already know a valid address).

After OpenATPSocket we transfer the new address into the top (excluding the nextEntry pointer) of myNtable. Then we pack3str to fill the rest of myNtable. Ablock.ntQEIPtr points to myNtable and after we specify an interval, a count, and set the verifyFlag we call RegisterName. Now, myNtable is the exclusive property of NBP, and it must not be disturbed before we RemoveName.

To prepare for GetRequest it is not necessary to set ATPsocket because it remains set from OpenATPSocket. Even though we don't expect a message in our message (it's existence is enough) we set Sblock.reqPointer and reqLength for the string buffer. Then we call GetRequest.

Three of the items that are listed on page 70 as being required for SendResponse are already set by GetRequest so those of you following along, don't be alarmed when we skip them. They are atpSocket, addrBlock, and transID. It is necessary to set the EOM bit of the ATPflags since this will be our last (and only) response. BdsPointer is set to myBDS. NumOfBufs and bdsSize are set to one because our reply string will easily fit into one packet and one BDSelement. MyBDS is set for the string reply and then we SendResponse.

The GetRequest - SendResponse loop is repeated until the mouse button is held down after a response. After that we RemoveName and CloseATPSkt. Note that unlike Bob Denny's version, this returns the same fortune every time. We leave it to you to jazz it up with a more varied response.

So, now we have our network. Happy talking!

### Get\_Fortune listing:

```
program Apple_talk_Access;{by Alan Wootton 7/85 }
type
 ptr = ^integer;
 strptr = ^str255;
```

```
Bsplit = packed array[0..1] of char;
```

```
AddrBlockRec = record
 aNet : integer;
 aNode_aSocket : Bsplit
end;
```

```
BDSElement = record
 buffSize : integer;
 buffPtr : ptr;
 datasize : integer;
 userbytes : longint
end;
```

```
BDSType = array[0..7] of BDSElement;
```

```
NtTuple = record
 NetworkNumber : integer;
 NodeID_SocketNumber : Bsplit;
 none_ObjectName : Bsplit;
 entityname : array[0..42] of integer;
end;
```

```
NamesTableEntry = record
 nextEntry : ^NamesTableEntry;
 NetworkNumber : integer;
 NodeID_SocketNumber : Bsplit;
 none_ObjectName : Bsplit;
 entityname : array[0..42] of integer;
end;
```

{ Parameter Block information, heavily modified for Appletalk}

```
ParamBlkPtr = ^ParamBlockRec;
ParamBlockRec = record
 qLink : Ptr;
 qType : integer;
 ioTrap : integer;
 ioCmdAddr : ptr;
 ioCompletion : ptr;
 reqTid : integer;
 ioNamePtr : ^str255;{ also UserData }
 ioVrefnum : integer;
 ioRefNum : integer;
 csCode : integer;
 case integer of
 0 : (
 ATPSocket_ATPFlags : Bsplit;
 AddrBlock : AddrBlockrec;
 ReqLength : integer;
 Reqpointer : ptr;
 BDSpointer : ^BDSelement;
```

```

numofBufs_timeoutVal : Bsplit;
numofResps_retrycount : Bsplit
);
1 : (
 curRBitmap_ATPflags : Bsplit;
 dummy1 : longint;
 confirmAddr : ptr;
 dummy2 : array[0..2] of integer;
 bitMap_BDSsize : Bsplit;
 transID : integer
);
2 : (
 interval_count : Bsplit;
 ntQEIPtr : ^namesTableEntry;
 verifyFlag_none : Bsplit;
 dummy3 : integer;
 newSocket_none : Bsplit;
 dummy4 : longint;
 rspNum_none : Bsplit
);
3 : (
 dummy5 : integer;
 entityPtr : ^char; { actually three packed str's }
 retBuffPtr : ptr;
 retbuffsize : integer;
 maxtoget : integer;
 numgotten : integer
)
end;

{common OS trap code}
{This nasty little piece is used to make the PBxx calls}
{described in the File Manager and the Device Manager}
{chapters of Inside Mac. This is covered in detail in }
{MacTutor vol.1 no. 6, and also no.7 }
function filecall (Pb : ParamBlkPtr;
 trap : integer) : integer; { OSErr }
var
 d0, a0 : longint;
 access : array[0..12] of integer;
begin
 stuffHex(@access,
'2848548C41FA000C309F245F265F20522013FFFF224826804
ED4');
 a0 := ord(pb);
 inlineP($4E75, @d0, @a0, trap, @access);
 filecall := loword(d0);
end;

{Given a pointer and three strings, pack them }
{end to end starting at the pointer }
procedure pack3str (strP : strptr;
 s1, s2, s3 : str255);
begin
 strP^ := s1;
 strP := pointer(ord(strP) + length(strP^) + 1);
 strP^ := s2;
 strP := pointer(ord(strP) + length(strP^) + 1);
 strP^ := s3;
end;

```

```

{Open the drivers '.MPP', and '.ATP' if they are }
{not already open, and if the serial port is free. }
{To understand this search high and low for }
{descriptions of PortBUse, and SPConfig, which }
{are system globals (I'm still not totally sure how }
{these work). }

```

```

function ATPLoad : integer; { OSErr }
type
 r = record
 use : char;
 end;
var
 pblock : ParamBlockRec;
 Tstr : str255;
 PortBUseP : ^r;
 SPConfigP : ^char;
 err : integer;
begin
 pBlock.ioNamePtr := @Tstr;
 pBlock.dummy5 := 0; { ioPermsn }
 PortBUseP := pointer($291);
 SPConfigP := pointer($1FB);
 with PortBUseP^ do
 begin
 writeln(' PortBuse is ', ord(use));
 if ord(use) > 127 then
 begin
 err := -98; { assume portNotCf }
 if (ord(SPConfigP^) mod 16) < 2 then
 begin
 Tstr := '.MPP';
 err := filecall(@pBlock, $A000); { open }
 end
 end
 else if (ord(use) mod 16) <> 1 then
 err := -97; { PortInUse }
 if (not odd((ord(use) div 16))) and (err = 0) then
 begin
 Tstr := '.ATP';
 err := filecall(@pBlock, $A000); { open }
 end;
 end; { of with }
 ATPLoad := err;
end; { of function }

```

```

function ATPcall (Pb : ParamBlkPtr) : integer;
begin
 Pb^.ioRefNum := -11;
 ATPcall := filecall(Pb, $A004); { control }
end;

```

```

function OpenATPSkt (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 254;
 OpenATPSkt := ATPcall(Pb);
end;

```

```

function CloseATPSkt (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 250;

```

```

CloseATPSkt := ATPcall(Pb);
end;

function SendRequest (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 255;
 SendRequest := ATPcall(Pb);
end;

function GetRequest (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 253;
 GetRequest := ATPcall(Pb);
end;

function SendResponse (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 252;
 SendResponse := ATPcall(Pb);
end;

function MPPcall (Pb : ParamBlkPtr) : integer;
begin
 Pb^.ioRefNum := -10;
 MPPcall := filecall(Pb, $A004);{ control }
end;

function RegisterName (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 253;
 RegisterName := MPPcall(Pb);
end;

function LookupName (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 251;
 LookupName := MPPcall(Pb);
end;

function RemoveName (Pb : ParamBlkPtr) : integer;
begin
 Pb^.csCode := 252;
 RemoveName := MPPcall(Pb);
end;
{ Above portion should be copied to Send_Fortune also.}

procedure Get_Fortune;
var
 Nblock, Sblock : ParamBlockRec;
 myNtable : NamesTableEntry;
 myTuple : ntTuple;
 myBDS : BDStype;
 strP : strptr;
 err : integer;
 reply : str255;
begin
 if ATPLoad = 0 then
 begin
 with Nblock do
 begin
 interval_count[0] := chr(1);

```

```

 interval_count[1] := chr(32);
 strP := pointer(ord(
 @myNtable.none_ObjectName[1]));
 pack3str(strP, '=', 'Dial-A-Fortune', '=');
 entityPtr := pointer(ord(
 @myNtable.none_ObjectName[1]));
 retBuffptr := pointer(ord(@myTuple));
 retBuffsize := sizeof(myTuple);
 maxToGet := 1;{ if larger use array of tuples}
 err := LookupName(@Nblock);
 writeln('lookup err', err);
 end;{ of with Nblock }
 if err = 0 then
 with Sblock do
 with myTuple do
 begin
 ATPsocket_ATPFlags[1] := chr(32);{atpXObit}
 addrBlock.aNet := networkNumber;
 addrBlock.aNode_Asocket[0] :=
 nodeID_SocketNumber[0];
 addrBlock.aNode_Asocket[1] :=
 nodeID_SocketNumber[1];
 reqLength := 0;{no request data}
 reqPointer := nil;
 bdsPointer := @myBDS;
 numOfBufs_timeoutval[0] := chr(1);{buffers}
 numOfBufs_timeoutval[1] := chr(2);{ seconds }
 numOfResps_retryCount[1] := chr(3);{ tries }
 myBDS[0].buffsize := 256;
 myBDS[0].buffPtr := pointer(ord(@reply));
 err := SendRequest(@Sblock);
 writeln('request err', err);
 writeln('fortune returned is - ', reply);
 end;
 end
 else
 writeln('Appletalk open error ', ATPLoad);
 end;
begin { main main main main main main }
showtext;
Get_Fortune;
end.

```

### Send\_Fortune listing:

**note:**upper portion is the same as Get\_Fortune.  
Copy and attach that portion to this.

```

procedure Send_Fortune;
var
 Ablock, Sblock : ParamBlockRec;
 myNtable : NamesTableEntry;
 myBDS : BDStype;
 strP : strptr;
 err : integer;
 buffer, reply : str255;
begin
 reply := 'You will live long and prosper';
 if ATPLoad = 0 then

```

```

begin
Sblock.addrBlock.Annet := 0;
Sblock.addrBlock.Anode_Asocket[0] := chr(0);
Sblock.addrBlock.Anode_Asocket[1] := chr(0);
Sblock.atpSocket_atpFlags[0] := chr(0);
err := OpenATPSkt(@Sblock);
writeln('open skt err', err);
writeln('socket',
ord(Sblock.atpSocket_atpFlags[0]));
myNtable.nodeID_SocketNumber[1] :=
Sblock.atpSocket_atpFlags[0];
strP :=
pointer(ord(@myNtable.none_ObjectName[1]));
pack3str(strP,
'MacPascal Fortunes', 'Dial-A-Fortune', '');
Ablock.ntQEIPtr := @myNtable;
Ablock.interval_count[0] := chr(1);
Ablock.interval_count[1] := chr(32);
Ablock.verifyFlag_none[0] := chr(255);
err := RegisterName(@Ablock);
writeln('register err', err);
if err = 0 then
begin
writeln('Sending fortunes, press mouse to quit');
repeat{ send fortunes }
begin
Sblock.reqPointer := pointer(ord(@buffer));
Sblock.reqLength := 256;
err := GetRequest(@Sblock);
writeln('get err', err);

```

```

Sblock.ATPsocket_ATPflags[1] := chr(16);
{ATPEomBit}
Sblock.bdsPointer := @myBDS;
Sblock.numofbuffs_timeoutval[0] := chr(1);
Sblock.bitmap_bdssize[1] := chr(1);
myBDS[0].buffPtr := pointer(ord(@reply));
myBDS[0].dataSize := length(reply);
myBDS[0].buffsize := 256;
err := SendResponse(@Sblock);
writeln('send err', err);
end
until button;
Ablock.entityPtr :=
pointer(ord(@myNtable.none_ObjectName[1]));
err := RemoveName(@Ablock);
writeln('remve err', err);
err := CloseATPSkt(@Sblock);
writeln('close SKT err', err);
end;
end
else
writeln('Appletalk open error ', ATPLoad);
end;

begin { main main main main }
showtext;
Send_Fortune;
end.

```



## For What It's Worth Dept.

Peter Wollschlaeger of Hildeseim, West Germany, sent in this little gem. Try it, you'll like it! Peter says "...the most information I ever got for a Mac programmer like me in Macworld was an ad to subscribe to MacTutor!" Peter also reports that trying to run the MDS Editor under Finder version 3.3 bombs. Update to 4.1 Peter. Then your only problem will be missing icons occasionally!

**program Bugged;**  
**var**

```

r : rect;
left, top, right, bottom : integer;
icon1, icon2, width : integer;

```

**begin**

```

left := 10;
top := 10;
right := 40;
bottom := 40;
width := right - left + 10;
icon1 := 1;
icon2 := 257;

```

```

showdrawing;
setrect(r, left, top, right, bottom);
ploticon(r, geticon(icon1));
setrect(r, left + width, top, right + width, bottom);
ploticon(r, geticon(icon2));

```

**end.**



# *Pascal Procedures*

## *The Amazing Pic to Clip Utility*



Alan Wootton  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 11

### Miscellaneous Month

This month there will not be a main topic - instead I have a collection of goodies that I have been saving up. We will talk a little about some books, about the compiler search I am conducting, revisit printing, and cover two more short topics.

Before I begin, I would like to talk a little philosophy. I wish to bare my soul on what Pascal means to me. There has been some talk about "fluff" in this publication, and this makes me wonder about Pascal. Is Pascal a Real Programmers language? Or is Pascal for Quiche Eaters? (See Feirstein, B., "Real Men don't Eat Quiche", New York, Pocket Books, 1982.) It was once thought that Real Programmers used Fortran. Today it would be C (and always assembly). In that perspective, Pascal is indeed for Quiche Eaters, and there are undoubtedly millions of Quiche Eating Pascal programmers. Let us not forget however, that the ROM was originally in Pascal, and MacPaint still is. These are not programs created by Quiche Eaters. It was a Real Programmer, who, in a fit of rebellion, created LinlineF and Generic. It was a Real Programmer who mangled Niklaus Wirth's (world-renowned Quiche Eater who created Pascal) specification by allowing for the beautiful expression `ptr:=pointer(ord(ptr)+$100)`. There is hope for Pascal.

Let's review what a high level language is for. In the dawn of time, there were only front panels with switches. Later, the Assembler was invented; the idea being to cater to the laziness of Real Programmers. This was acceptable because every conceivable possibility was still available. Afterwards, someone noticed that much of a programmer's work was making sure that parameters were passed correctly between procedures, and that the stack was not messed up. The high level language was invented to ease the drudgery of these tasks. Now those who used Fortran, the first HLL, weren't trying to get away from assembler so much as they were being lazy (one of the higher instincts of Real Programmers). And for those things not allowed in the HLL, assembly code could be linked in. All went well, except for the increasing numbers of Quiche Eaters who jumped straight into Fortran, bypassing knowledge of assembly language. Here is where the trouble begins.

By this time programming had become Real Big Business, and certain managerial types were troubled over the fact that their Quiche Eating programmers couldn't decipher the strange things the Real Programmers were now doing in Fortran (list processing, records, pointers, string manipulation, etc.). The

word was passed around that Fortran must be upgraded to allow for these strange new constructs. Along came Wirth to save the day, and the Quiche Eaters gained new momentum. What we had now was a deluge of programmers, working in a toy operating system, who had little or no knowledge of how a computer really works. Real Programmers were in danger of becoming extinct. But, let us not forget that the true purpose of a HLL is as a shortcut to machine code for lazy Real Programmers. As Pascal became popular, extensions were made to the language so that Real Programmers could use it. Still, Pascal was increasingly distant from the workings of the machine it ran on (Basic and microcomputers fuel this trend). This accounts for the popularity of Forth, a language which wallows in the characteristics of its processor and is infinitely extendable and unreadable (perfect for a Real Programmer).

The 68000 changes all this. The 68000 is designed with an instruction set that provides for the efficient implementation of most HLL statements. On the Mac, you can trace machine code and see it follow, line by line, the structure of a Pascal program (I do this all the time). At last Real Programmers have a language that raises laziness to a high art. And MacPascal is at the top of the heap. One jots down the code he (she) wants, and it is immediately checked for syntax, semantic form, and even emulated to check for logical errors. It can then be automatically and effortlessly converted into machine code. Or can it?

The Real Programmers at Apple are a snotty bunch (supposedly the worst were canned, but I don't believe it). They think that all the Real Programmers in the world work for Apple, or Microsoft, or IBM (blue suits), or NASA (rocket jocks), or the National Security Agency (spooks), and that there is no need to provide a compiler for the Quiche Eating public. You see they all have compilers on their Lisas (so do I, but I hate it). Therefore, Mr. Jobs allowed Apple to shoot itself in the foot, and there is *still* no compiler. Now I will admit that MacPascal is perfect for the Real-Programmer-growing-up and for the Quiche Eaters too. But Real Programmers cannot exist for long on MacPascal and UCSD Pascal. They need a native code compiler or they will all switch to the excellent C systems available for the IBM PC (interpreter, preprocessor, compiler, optimizer, runtime profiler, assembler, libraries, source code debuggers, all integrated into one system). It would be a shame if the Mac died for lack of Real Programmers.

For me this is the real purpose of Pascal: a lazy way to write machine code. Like a spelling checker or a business



letter generator, Pascal is a way to save time, not an excuse to not learn to spell or write. One can use MacPascal to diddle with the system, knowing full well that his efforts can be smoothly integrated into the natural Mac way of programming. This argument should explain why an interpreted language, or a pseudo code system won't do. Not to mention the fact that without native code you can never make desk accessories, window definition functions, control definition functions, INIT resources, etc. More importantly, if you compile your procedures into resources (the standard method), you can call them from inside MacPascal. (For an example of this see the article on printing, August Vol.1 No.9, where the PDEF resources are called in just this manner. This way when your MacPascal program gets too long, you just compile parts of it until it is more manageable. If it weren't for MacPascal I would switch to C in a second.)

### The Great Compiler Hunt

First of all, Apple intends to port the excellent Lisa compiler over to the Mac (without the dumb OS it now uses). The problem is that they don't have the good sense to just do it - they are going to rewrite the whole MDS system. I was told it would not see pre-release until December. My personal prediction is that we will all grow old first and then we will have to buy the new Mac and the new hard disk, to the tune of about \$5000.

[Note: MacTutor is pushing for a ".REL" file standard that would allow linking of object code files from a variety of programming languages. We are on record as supporting the MDS ".REL" file format as that standard with proposed extensions submitted by Bill Duvall's new linker, which is compatible with the MDS system as well as with his own C system. MacFortran from Microsoft is also very close to being compatible with this standard, and hence Bill's new linker. We encourage Apple and other proposed compilers to support the MDS ".REL" file format for object code files so that all languages can be linked with one common linker. We hope that Apple, in developing the Lisa compiler for Pascal on the Mac, will consider conforming to this standard. What we don't need is for Apple (and others) to give us yet another incompatible object code file format (i.e. don't dump the MDS system Apple!) [More on this in upcoming issues. -Ed.]

Tom Leonard of TML systems, Melbourne, FL. (a Real Programmer for sure) is writing a Pascal compiler to go with the MDS system (by Bill Duvall, another Real Programmer). It was going to be in beta test by the writing of this article but didn't make it. Next month, look for a Desk Accessory in this column. It will be emulated in MacPascal and compiled in TML Pascal. If this comes true we can do great things. He plans to sell it for \$49.95 Stay tuned..

I have a beta copy of Rascal by MetaResearch of Portland OR. This is a very interesting package. They wrote it to do real time I/O and it is great for that. It runs under an application that integrates editor, compiler, linker and runtime

package. What you do is write procedures named `_Init`, `_Main`, `_Mouse`, `_Key`, `_Update`, `_Event`, `_Menu`, `_` and `_Halt`, and the system calls the correct procedure as needed. The code (native 68000 code) is put into the data fork of the object file with a jump table leading to the procedures. Provisions are made to create standalone applications (with a 19K overhead). If you wrote an application to break the data fork out into resources you could make anything you wanted. The bad news is that it is not quite Pascal. There are no declared types, and other less important parts are not implemented. The upgrade promises to correct some of these, but I haven't seen it (even though they said it would be ready long ago). The other problem is that it is copy protected (although Steve Jasik's NOSY method of getting things onto hard disks works well). You can buy it now for \$99.

I heard rumors that Borland, of Turbo Pascal (for the PC) fame, was doing a compiler for the Mac. Where it is, I don't know. The other bad part is that it seems Turbo Pascal is not native 8086 code. I would not count on them producing native 68000 code either. Many of the Amiga announcements promised Pascal from Borland, so we shall soon see.

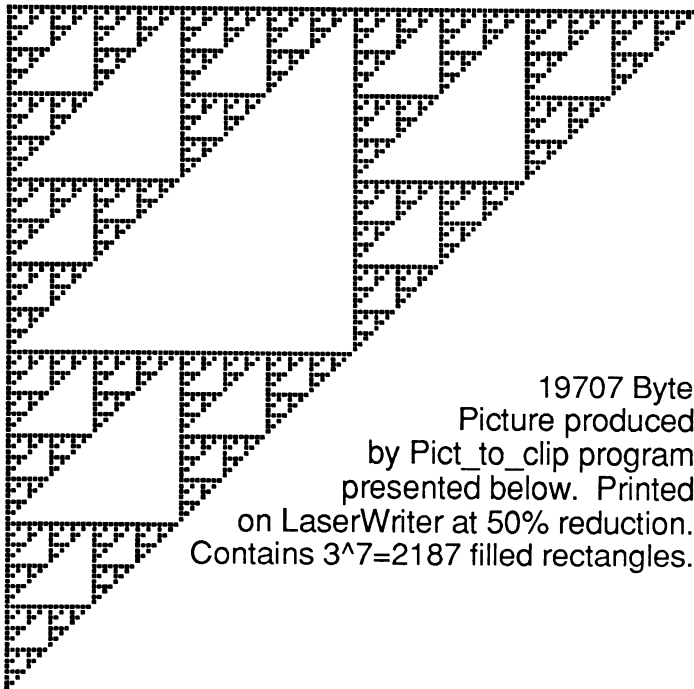
UCSD has had a Mac compiler for some time. It makes P-code so I don't know how you are supposed to debug it. You can make standalone applications. But you can never call procedures in resources from MacPascal with this system. If it weren't so un-Mac-like in style, this would be the premier Quiche Eaters system.

### Books

I have three interesting new books. Two on MacPascal, one on programming the Mac. The MacPascal books are The MacPascal Book, by Paul Goodman and Alan Zeldin, Brady Communications company, and Macintosh Pascal, by Lowell A. Carmony and Robert L. Holliday, Computer Science Press. Both are mostly about learning Pascal with MacPascal. Both are also filled with excellent examples, something that the MacPascal docs are short on. If you are a Real-Programmer-in-Training you should get one of these and do some of the examples. It is hard for me to do a good review for two reasons: 1. I already know Pascal; 2. I am familiar with the MacPascal system. The true test is whether a beginning Pascal person can get up to speed using only one of these books. I was told by a computer store person that these books would get yanked from circulation because with one of these books and a pirate copy of MacPascal, you have no reason to buy the real thing, and Apple doesn't like that. If this is true, then they are good books indeed (although you should still buy the product, the MacPascal creators deserve the money). See for yourself. If anyone has any experiences, write a review for MacTutor. I'm sure Dave Smith (our illustrious publisher) would be interested. The main differences between them are that The MacPascal Book is typeset, while Macintosh Pascal is in Geneva 12. The MacPascal Book also has appendices covering useful things including answers to exercises, syntax diagrams, Quickdraw and SANE.

I also have a copy of Macintosh Revealed by Stephen Chernicoff, Apple Press, Hayden Book Company. This is the first true alternative to Inside Macintosh. The true test is whether one can learn about the Mac using this book, and without Inside Mac. I would guess that it passes. Even if you already have Inside Mac, there are interesting things here. Like the formats of all finder-related resources, for instance. Chernicoff is from Apple, so the book is very accurate. I was shocked, though, to see that he forgot the register information for PACK 7, which his book implies is strictly stack-based. He says the information is for all languages, although the examples are for Lisa Pascal. (When is Apple going to finally give up on Lisa Pascal?).

Enough fluff! I have some programs to present.



19707 Byte  
Picture produced  
by Pict\_to\_clip program  
presented below. Printed  
on LaserWriter at 50% reduction.  
Contains  $3^7=2187$  filled rectangles.

### Printing Revisited

After I wrote the article on printing in August I realized people might just want to make a printout of a large graphic using MacPascal without having to deal with the details of the Printing Manager. The way to do this is to run Pict\_to\_clip (below), quit MacPascal, start MacDraw, paste, and print.

What Pict\_to\_clip does is save the quickdraw calls that you make in the form of a 'picture'. Pictures are covered in the MacPascal Appendix on quickdraw, so I won't go too deep. All you have to do is OpenPicture, then make some quickdraw calls and then ClosePicture. You may then DrawPicture, if you wish, and when you are done you KillPicture to free the memory used by the handle. Once you have a handle to your picture, you want the program to copy it onto the clipboard. After it is on the clipboard you can paste

it in the scrapbook, or just go directly to MacDraw and manipulate and print it all you want.

Clipboard is the name of the file that the scrap is saved in on USA Macs. On other Macs it has different names. So what we are really dealing with here is the Scrap Manager. The Scrap Manager has its own chapter in Inside Mac, so we should all refer there now (even though we'll only use two functions). First we ZeroScrap; very simple. Then we PutScrap using a pointer to the picture, its length and a longint that is the four ascii chars 'PICT'.

DrawSomething is the procedure that makes the quickdraw calls. The first example is a recursive routine that draws the fractal above. The second draws a  $\sin(x)/x$  function. If you need to know more about recursion and fractals, get one of the books above, as both cover the subject. The  $\sin(x)/x$  function is an exercise for the reader. (Note that the program draws the picture upside down. Why?). The limitations DrawSomething must observe are that if the PICT gets greater than 32K, the Mac locks up. Furthermore, if you paste a 30K picture into MacDraw, it gets very slow (2 minutes to update the screen). I have made more intricate drawings by using the LaserWriter with the program from August (I must have done something right 'cause it worked). This won't work with the ImageWriter, because there is a 32K/page limit on spool files. More after this short program.

Program to put a picture onto the clipboard. DrawSomething procedure could be anything. This example draws the fractal shown above.

```
program Pict_to_clip;
uses
 sane, Quickdraw2;

procedure DrawSomething;
const
 stopsize = 4;

procedure SquareTo (xxx : integer);
var
 pt : point;
 r : rect;
begin
 getpen(pt);
 if xxx > stopSize then
 begin
 xxx := xxx div 2;
 moveto(pt.h + xxx, pt.v);
 SquareTo(xxx);{ upper right sub square }
 moveto(pt.h, pt.v + xxx);
 SquareTo(xxx);{ lower left sub square }
 moveto(pt.h, pt.v);
 SquareTo(xxx);{ upper left sub square }
 end
 else
 begin
 setrect(R, pt.h, pt.v, pt.h + xxx, pt.v + xxx);
```

```

 PaintRect(r);
end;
end;

begin { of DrawSomething }
 moveto(-256, -256);
 SquareTo(512);
end;

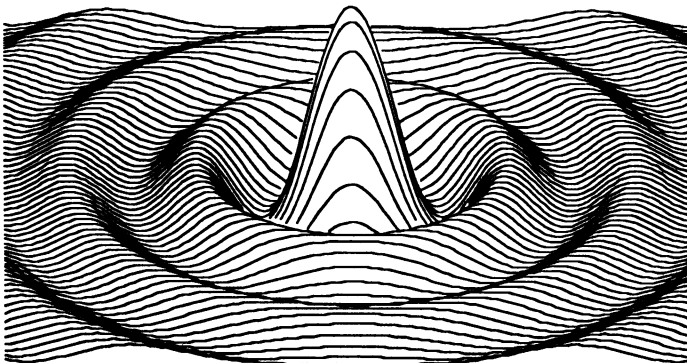
procedure main;
var
 bigR, normR : rect;
 PicH : PicHandle;
 length, theType, Ill : longint;
begin
 setrect(bigR, -1000, -1000, 1000, 1000);
 setrect(normR, 0, 0, 250, 250);
 clipRect(bigR);

 picH := OpenPicture(bigR); { 2000 by 2000 plotting area }
 DrawSomething;
 ClosePicture;
 DrawPicture(PicH, normR); { compress 2000^2 into 250^2 }

 Ill := LinlineF($A9FC); { _ZeroScrap }
 Hlock(PicH);
 theType := $50494354; { 'PICT' }
 length := GetHandleSize(PicH);
 writeln('length is', length);
 Ill := LinlineF($A9FE, length, theType, picH^);
 { _PutScrap }
 KillPicture(PicH); { also disposes of handle }
end;

begin
 showDrawing;
 main;
end.

```



21026 Byte Picture produced by changing the DrawSomething procedure in Pict\_to\_clip. Printed on laserWriter at 50% reduction. Contains 8192 line segments.

DrawSomething procedure to produce second drawing shown above. Use this code in Pict to clip.

```

function form (x, y : integer) : integer;
var
 d : extended;
 y2, x2 : integer;
begin
 x2 := x - 257;
 y2 := y - 257; { 257, 257 is center peak }
 d := sqrt((x2 * x2) + (y2 * y2)); { d is distance from center }
 d := sin(d / 15) * 2000 / d;
 form := num2integer(d) + (y div 2) + 50;
end;

procedure DrawSomething;
const
 yst = 8;
 xst = 4;
var
 hi : boolean;
 V, x, y : integer;
 arr : array[0..2000] of integer;
begin
 y := 0;
 while y < 512 do
 begin
 x := xst;
 hi := true; { above last line }
 moveto(0, form(0, y));
 while x < 512 do
 begin
 V := form(x, y); { calculate function }
 if V < arr[x] then
 begin { if new value under previous line }
 moveto(x, V);
 hi := false;
 end
 else
 begin { new value above previous line }
 if hi then { were above before, no change }
 lineto(x, V)
 else { changing from under to above }
 moveto(x, V);
 hi := true;
 arr[x] := V;
 end;
 x := x + xst;
 end;
 y := y + yst;
 end;
 end;
 end;
end;

```

### Random Problems

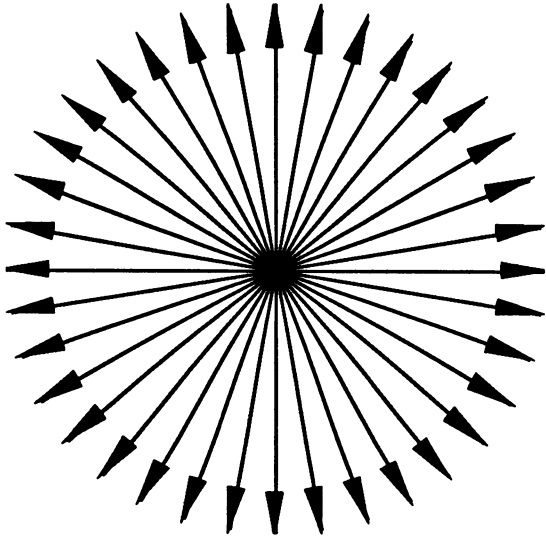
Ed Groth of Scottsdale, Az. brought to my attention a bug in MacPascal. It seems that Quickdraw and SANE both have a random function with the same name! The SANE Random is therefore unavailable. The fix is to wade through the SANE chapter in Inside Mac until you discover that all of SANE is in packages 4 and 5, and that you can call the functions directly. Use the following incantation.

```

inlineP($A9EC, @x, $20);
{ _elems68K, $20 is Random }

```

This performs the function `x:=random(x)`; Use any extended type number instead of `x` in your program.



### Arrowheads

I liked Rick Flott's discovery of how to draw arrowheads so much (Toolbox Tips, MacTutor Vol.1 No.9) that I had to do it myself. For those who can't decipher C, I present my version below. For a complete description of how it works refer to the August issue. To make the drawing above, I used `Pict_to_clip` to make a 200x200 picture that I pasted directly into MacWrite. Note that it looks much better here than it does on the screen because of the LaserWriter. If it were a MacPaint cutout instead of a PICT, it would look the same as it does on the screen.

Procedure to draw arrow heads. Use this in any program just as you would `Lineto`. Note: no example of its usage is included here.

```
{ ArrowLineto works just like Lineto except }
{ an arrow head is drawn at the point H,V. }
{ The size and width of the arrow head are }
{ determined by the constants. }
procedure ArrowLineto (H, V : integer);
const
 Asize = 20; { angular size of head }
 Ah = 10; { always half of head angle }
 aL = 16; { length of head }
var
 startpt : point;
 R : rect;
 Ang : integer;
begin
 setrect(R, H - aL, V - aL, H + aL, V + aL);
 getpen(startpt);
 PtToAngle(R, startpt, Ang);
 lineto(h, v);
 paintarc(R, Ang - Ah, Asize);
end;
```

### Out of Memory?

If you are getting the message "not enough memory to continue execution, hiding the program window may help", when you know full well there is plenty of memory, it is because a nonrelocatable block has been allocated on the heap after execution began. Opening windows or dialog boxes is the most common source of this problem. If your final code is to be a desk accessory, you will have this same problem with Microsoft products. The solution is to call `NewPtr`(small amount), open your windows, and then call `DisposePtr`. This will leave a little space under your window for MacPascal (or Microsoft Chart) to use when it calls the poison function `SetPtrSize`. If you are an advanced enough user to get this problem then you are advanced enough to know what I mean.

I think this covers it for this month, so, see ya next time. And if you have problems or questions or suggestions then write me care of MacTutor!



# Pascal Procedures

## A Resource Utility DA with TML Pascal



Alan Wootton  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 12

The subject this month is a sample Desk Accessory, but the real news of note is the emergence of a Pascal compiler for the Mac. The Desk Accessory (DA) presented here will be compiled with TML Pascal and is installed using Apple's Font/DA Mover program. The DA is called ShowRes and is used to view the resources that are open on the heap. Between the compiler, the desk accessory, and the subject of resources there is more than can be presented in one month. The strategy will be to discuss the overall structure of DAs, how to compile, and details on resources. We'll put off until later the complete exposition of desk accessories. In order to show the overall structure of a DA, I have a MacPascal program that will emulate the system's treatment of DAs. I will present this program next month.

### How to Compile

The compiler is not a simple, one-application system, like MacPascal is. The TML Pascal compiler works with what I call the MDS system. This means it is compatible with the Assembler, Linker, and Editor sold by Apple. As such, there are several applications you run before a completed program is produced. First there is the editor. The editor is the application EDIT that edits files of type TEXT. This same editor is provided with several other programs, and you can also use any editor that will create TEXT type files. MacWrite for instance. With the editor you create the Pascal source code, and the Rmaker source (if needed). In figure 1 I have provided a diagram of what steps you might use in two different situations.

The next stage is the compiler. The compiler will produce assembly language, if desired, but for convenience you may skip the assembly phase and make .rel files directly (.rel is the assembler's normal output). The compiler comes with everything except the assembler, so, if you want to assemble you must get the assembler from Apple. The Consulair C compiler can also be used as an assembler in this system (Consulair also has a more professional linker that you can use). After you have converted your code to .rel files you invoke the linker. The linker's normal output is a file with CODE type resources in it. Other resources may be copied in from another file if desired. Depending upon your situation, you may be done. If not, Rmaker is used to create resources for use by the program (there are other ways also). If you are writing a DA, you use Rmaker to create your DLOG and other resources, and also to copy the CODE 1 resource from the linker output file and convert it into a DRVr resource.

### Compilation steps for typical programs

| typical Application        | typical Desk Accessory |
|----------------------------|------------------------|
| EDIT                       | EDIT                   |
| prog.pas                   | da.pas                 |
| prog.R                     | da.R                   |
| RMAKER                     | COMPILE                |
| prog.R } only when changed | da.pas                 |
| COMPILE                    | ASM                    |
| prog.pas } optional        | da.asm                 |
| ASM                        | LINK                   |
| prog.asm } optional        | da.link                |
| LINK                       | RMAKER                 |
| prog.link                  | da.R                   |
| test program and repeat    | Font/DA Mover          |
|                            | da.acc                 |
|                            | test da and repeat     |

note: There are hundreds of variations possible. This is only a sample.

Figure 1: Compilation Steps

With six programs, involved I can't very well describe the detailed operation of each individually. So I will make it brief. Compiling ShowRes consists of following the sequence illustrated in figure 1 for DAs. Use the files ShowRes.pas, and ShowRes.R. Both are listed below. There is an option on one of the compiler menus that you choose if you are making a DA and not an application. Don't forget it, or you will end up with the wrong kind of code. Figure 2 shows all the files involved in creating ShowRes.

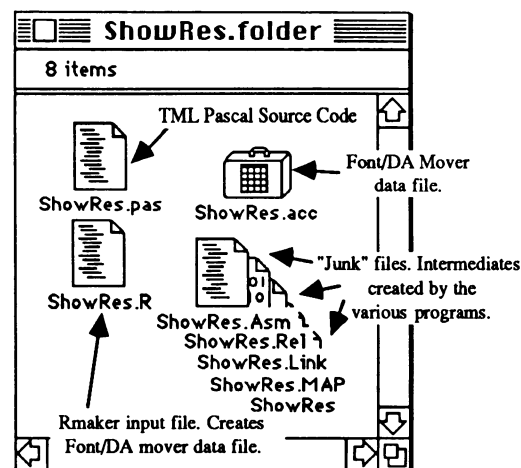


Figure 2: Files used in creation of ShowRes

When you invoke the DA option of the compiler it causes your program to be linked with a header that is in assembly language. The structure of this header matches what is described in Inside Mac for the structure of DAs in assembly language (Inside Mac does not even hint that DAs can be done in Pascal). This DA header is shown below:

```

;-----
;---File AccHead.asm
;-----
;---This is a generic Desk Accessory
;---interface for Pascal.
;-----
;---It is linked with a .rel module
;---containing the three xdefs
;---Open, Ctl, and Close. These must
;---take two pointers on the stack.
;-----
;---written 9/85 by Alan Wootton.
;-----
INCLUDE MDS:SysEquX.D
INCLUDE MDS:ToolEquX.D
INCLUDE MDS:QuickEquX.D
INCLUDE MDS:FSEqu.D
INCLUDE MDS:MacTraps.D

; declare the three Pascal procedures
; that are used for a Desk Accessory

xref open
xref close
xref ctl

; Driver flags and information
; those below should fit most needs
;-----
;Flags are:
; 1- dNeedTime: receive periodic service
; 2- dCtlEnable: can take control calls
;All other flags are not set.
;
; service rate is one call every second.
; menuID is filled by open if used

dc.w $2400 ; Flags
dc.w $003C ; service rate
dc.w $FFFF ; event mask
dc.w $0000 ; MenuID goes here

; Entry point offset table

dc.w OrnOpen ; open routine
dc.w OrnDone ; prime
dc.w OrnCtl ; control
dc.w OrnDone ; status
dc.w OrnClose ; close

OrnOpen
moveM.l a0/a1,-(sp) ;save regs
move.l a1,-(sp) ;stack Device

```

```

move.l a0,-(sp) ;stack Param

jsr Open ;visit Pascal land
bra.s orndone

OrnClose
moveM.l a0/a1,-(sp) ;save regs
move.l a1,-(sp) ;stack Device
move.l a0,-(sp) ;stack Param

jsr Close ;visit Pascal land
bra.s orndone

Orndone
moveM.l (sp)+,a0/a1 ;restore regs
clr.l d0
rts

OrnCtl
moveM.l a0/a1,-(sp) ;save regs
move.l a1,-(sp) ;stack Device
move.l a0,-(sp) ;stack Param

jsr Ctl ;visit Pascal land

moveM.l (sp)+,a0/a1 ;restore regs
move.l jIODone,-(sp)
rts

end ;of assembly
;-----
;-end of file
;-----

```

Don't be put off by the many steps described to make a program. If all you want is a "plain vanilla" Pascal program, then there is a compiler option that will open a window, initialize the system for you, and allow regular readln and writeln in that window. You may also draw in the plain vanilla window if you wish. The whole bit with Rmaker is just for those who wish to make a Mac-style program with menus and windows and everything. Most simple MacPascal programs can be compiled with very little modification this plain vanilla way.

### What a Desk Accessory Is

If you have a Mac and you are reading this publication, you undoubtedly know what a DA looks like. You may not be aware of what its structure as a program is. The main difference between a DA and an application, from a programming standpoint, is that a DA has *no* main procedure. There is no main loop that obtains events, and there are no global variables. It is easiest if you imagine that you have written a standard program shell that does *some* of the parts of a program and that your task is to complete three routines to do the rest. The three routines are Open, Close and Ctl (short for Control). The system will call Open whenever the DA is selected from the Apple menu, and it will call Close when the

close box on the window is clicked, or when someone calls `CloseDeskAcc` (see desk manager chapter of *Inside Mac*). Between these two the system will call `Ctl` to enable your DA to handle certain occasions.

Two records are passed to DA routines by the system. They are the Parameter Block Record and the Device Control Entry (DCE). The Parameter Block is the same record used in file manager calls and device manager calls (a DA is a type of driver). A new parameter block is formed, by the system, for each call the DA receives. Its main use is the `csCode` field, which indicates the purpose of the control calls. Some values of `csCode` are predefined (and are declared in the program), others are available for your use (if you wanted to communicate between DAs, for instance). The Device Control Entry is created by the system for it to use to keep track of your DA. Certain of the fields are available for your use. `DctlStorage`, `dctlWindow`, `dctlEmask`, and `dctlMenu` are safe to modify. The others are for use by the system. I have redeclared the DCE record in `ShowRes` because it is convenient for `dctlStorage` to be defined as a handle to my data and not simply as a handle to a byte.

Besides the issue of control, there is the problem of data. A DA has no global data. You cannot simply say `VAR` and then list the variables you are going to use, they would get destroyed at the end of that procedure. The system does, however, provide a facility for you. The Device Control Entry. You use this table as your *only* source of permanent storage. It is mandatory to put your window pointer into `dctlWindow` and you may use `dctlStorage` as a handle to additional data. Once you get the idea of what's going on you will realize that there are a great many pathological conditions to be dealt with. How much is done by the system, and how much is left for the DA? For instance, do you need to ever call `DragWindow`? If you look at the code below you will see many strange things. Why is the `windowKind` field set? I could cover some of these here. If I did then this article would be twice the length it is, so, I'm going to try to list everything there is to know next month. You should be able to make simple DAs by following the example given, at least for a while.

One issue won't wait. A DA is installed in the system file, and after it is installed it no longer has a file or an icon. So, where are its resources? A DAs resources are transferred along with it into the system file it resides in. In order to prevent collisions among the various DAs, Apple has come up with a funky scheme for numbering the resources owned by DAs. "All DAs will have 32 resource IDs allotted to them. The IDs for DAs will begin at `$C000` and number upwards." This means that if you are `DRVr 16` then your resource numbers will begin at `$C000+32*16`. All resources outside this range will not be transferred by the Font/DA Mover! Note that if the font/DA Mover decides to change the number of your DA it will also change the numbers of all your resources. So, you must calculate the ID of your resources at

run time. This calculation is performed in the open routine of `ShowRes` and the result stored in the `dctlMenu` field of the DCE. Note that the `dctlRefNum` is not exactly the ID of the driver, to convert use the formula `ID:=-1*(1+dctlRefNum)`.

The font/DA Mover tries hard, but does an imperfect job of keeping your resources in correct reference to each other. It changes the part of a DLOG resource that refers to a DITL but when renumbering MENU types it does not change the `menuID` field of the menu record. This is the purpose of the patch you see in the `Activate` procedure of `ShowRes`. Apple claims that DLOG and ALRT references to DITLs are renumbered correctly, and that DITL references to ICON, PICT, and CTRL are right. References by MENUs to MDEFs are supposed to work, but you can bet all other cross references are left incorrect. Your average DA will get by just fine using only dialogs and menus, so let's leave these problems to advanced users.

### Resource Rundown

Cruising through *Inside Macintosh*, one is likely to get the impression that resource files are some mysterious place that window and menu definitions go. The thing to remember is that resource files are *files*. It is helpful to imagine what you would do if you were writing a program that maintained 100 or 200 pieces of information of all different sizes. What would you do? You could write the data into a file with length words and then traverse the data like a list of strings. In addition to the length, you might include some identifying information. Another way would be to have an array of offsets into the file, and to use this to find individual items. Eventually you will find the need to read and write single pieces of the data without traversing, or loading, the whole file. In general you would want a complete and bullet proof collection of routines to handle these functions. Fortunately Apple has provided a set of traps to handle these kind of things for us.

What Apple has done is to arrange references to the pieces of data into a "map" at the end of the file. The map contains information relating to the file manager access path for the file being used, as well as a complete list of the data pieces, or resources, contained in the file. As an added bonus, the data is broken into broad categories called "types" and sub-categories referred to by using an "ID". It is as if Apple created a whole filing system contained within a file (just as the file manager maintains files in a volume). The pieces of data can be recalled by the order they were put in, ie. "give me the 5th resource". Use `GetIndResource` for this. Or, the data items have IDs so you can recall them by ID match, "give me the resource with ID #1234". Use `GetResource` for this. As an example, when you open a dialog with ID #257, the dialog manager makes the call `GetResource('DLOG',257)` to get the information it needs.

When the file manager creates a file, it initializes two lists of disk space for that file. Either of these lists can be

empty (or both! it is possible to have files that occupy 0 K of disk space). The nomenclature "Data fork" and "Resource fork" are used to refer to these two lists. When the resource fork of a file is opened (by calling `OpenResFile`) the map is read into memory. A field in the map known as the "resource file reference number" is set to the `ioRefNum` of the open file. The maps of all open resource files are arranged in a list in memory. New files opened are added to the head of the list. One may call `GetResource` or `GetNamedResource` to retrieve the individual resources. Beware of the fact that since all open maps are in the list, the resource manager will return the first resource it finds that fits the description of what it is looking for. It is possible to avoid searching some of the maps at the head of the list, but in general it is difficult to search only one map. In order to survey all the resources in all the maps a function named `GetIndResource` is provided. A count of the total number of resources in all the maps may be obtained by using `CountTypes`, and `CountResources`. There are also routines for adding resources and modifying their status. The function of the ShowRes desk accessory is to display the contents of the maps.

In the maps, besides the descriptive information of a resources type, ID, name, and attributes, is a handle to the resource data. Many of the resource routines use the handle to refer, unambiguously, to the resource in question. You must use `GetResource`, `GetNamedResource`, or `GetIndResource` to obtain the handle. The resource map contains a space to store the handle, and this space can be in one of three possible conditions. First, when the map is first opened the space is zeroed out, ie. there is no handle. Later, if the resource is referred to, a handle is obtained from the memory manager. The handle may simply be allocated out of the memory manager's list of master pointers without actually loading the data from the file. At this point the the map, and anyone else who asks, has a handle which points to a master pointer. However, unless the data is actually loaded, the master pointer is zero and there is no space allocated on the heap. If the data is then loaded, heap space is allocated, and the master pointer is set to point to the data on the heap. As you can see, the memory manager and the resource manager enjoy a very symbiotic relationship. We don't have time to discuss the memory manager in full here, but let me note that the resource manager uses some of the advanced features of both the memory manager, and also the file manager. Note that you may refer to a resource using it's handle, even if it occupies no space on the heap.

### The Desk Accessory ShowRes

I created ShowRes to be an example DA. Rather than just do a 'shell' I decided to give it a useful function. The function is to peek at the resources that are available while an application is running. Many times I want to just see if a resource exists that is supposed to. Or, I need to find the length of a particular resource, or find where a piece of code begins on the heap. It is a hassle to return to the finder for such simple tasks. It also a hassle to read the maps in

### Structure of ShowRes Desk Accessory

```

//
procedure Open(-,-);
begin
 Get New Dialog box
 set windowKind:=dctlRefNum
 Get handle for storage
 initialize storage
end;
//
Procedure Close(-,-);
begin
 delete menu
 dispose storage
 dispose dialog box
 clear dctlWindow
end;
//
procedure Ctl(-,-)
begin
 setport to self
 lock storage
 case csCode of
 accEvent
 case event.what of
 mouse down
 case dialog item of
 close button
 forget some of
 what we know
 view rect
 find what was clicked on
 add to what we know
 force update
 1 of 2
 'click'
 }
 }
 1 of 3
 'draw'
 }
 procs
 update
 draw what we know
 activate
 insert or delete menu
 accCursor
 if in view rect then
 set cursor to cross
 accMenu
 case menu item of
 about
 show about dialog
 loadResource
 if there is a handle then
 load it
 releaseResource
 if handle is loaded then
 release it
 otherwise nothing
 unlock storage
 end;
 //

```

separate procedure

Figure 3: ShowRes overall structure



memory, in hex, with a debugger. Hence, ShowRes. The way ShowRes works is that it maintains variables for; a resource type: TheType; for an ID:TheID; and for a resource handle:TheRes. If all these variables are empty (set to 0) then a list of all the types is drawn in the window. If TheType is set then a list of all the ids of that type are drawn. If TheType, and TheRes are set then the information for that resource is drawn. Clicks in the information drawn are interpreted as a selection of that type (or ID) and the information about 'what is known' is updated accordingly. To reselect a type or ID you use a button that will reduce 'what is known' (ie. if just TheType is known, clear it, if TheType and TheRes are known, just clear TheRes). The worst part of the code is the calculation of which type or ID a click has landed on. Since the information is drawn in a grid in the window, all it takes is some div's and mod's to find where in the grid the click occurred. Once you know which cell in the grid was clicked on, you can find which type or ID it was. It can be hard to get this kind of stuff right the first time so I worked with it in MacPascal until I had it right (it would have been better to use a box with scroll bars for this, but that would have taken a lot more code). To see the overall structure of ShowRes see figure 3.

### MiniEdit

How would you like to have the source code to a *complete* Macintosh application? An application with multiple windows, a full menu bar, desk accessory support, the works. It would be a good place to start on your project wouldn't it? How about if this code were in Pascal and you could modify and recompile it at will? Further, imagine that this code is very modularly written and fully commented. Interested? Now, what if in addition to the comments in the code there were an excellently written, and comprehensive, document that described the detailed function of every procedure? I'm not talking about twenty or fifty pages of programmer's notes. I'm talking about eleven *hundred* pages of carefully prepared and cross referenced material. In addition to the program description, you get a description of every toolbox routine used (and perhaps some that aren't).

Well, you can have it! It's called **Macintosh Revealed** by Stephen Chernicoff. You buy both volumes. The first one is basic toolbox calls, and the second volume is more advanced toolbox stuff and a description of the program **MiniEdit**. You then order the source code from Hayden Books for \$19.95. That's it. Watch this column for my adventures while compiling MiniEdit with TML Pascal (it is actually written in Lisa Pascal, but that should be no problem).

The nice thing about Pascal is that there are examples like this. TML Pascal will also compile the many program examples provided to developers by Apple (with slight modification).

## MacPascal and Inlines

If you have been following this column, you may have noticed that it is difficult to do very many Mac-like things in MacPascal without resorting to one of the inline statements (inlineP for procedures, BinlineF for functions returning booleans, WinlineF for those returning integers, and LinlineF for those returning 32 bit results). On the one hand, this is a blessing, since it enables one to hack around with the toolbox from within a very friendly environment. On the other hand I have come to the conclusion that unless you have a machine language debugger and know how to use it, inline will eventually give you a great deal of trouble. With this in mind, it becomes very difficult to describe how inline works. If I lapse into 68000 terminology then it is easy, but trivial. If I stick to Pascal then all I can say is that inline will invoke a toolbox trap in the place of a procedure or function. This is not strictly accurate though. What really happens is that the first word of the inline statement is actually executed by the processor, and the rest is put on the stack. If the first word is \$Axxx, where xxx is any three hex digits, then an A-trap is executed. All the A-traps are toolbox routines. If the first word is something else then anything could happen. For instance theStack:=LinlineF(\$2E8F) will return the position of the hardware stack, no toolbox traps are involved. The reason for this is that \$2E8F is the code for move.l sp,sp) in assembly language. With this capability it is possible to do all sorts of weird things. For instance, this is how I call the routine "access" I have used in previous articles. I am not particularly proud of this. It looks real ugly in the code. Most languages have the capability for this sort of hacking, but usually it is hidden inside some interface files and you don't have to know how it works. The better way to use MacPascal would be to make declarations like:

```
Function GetNewWindow (windowID: INTEGER; wStorage:Ptr;
 behind:WindowPtr):WindowPtr;
begin
 GetNewWindow := Pointer(LinlineF($A9BD ,
 WindowID , wStorage , behind));
end;
```

When you wanted a window you would just say GetNewWindow, with no inline to mess with. This also makes conversion to TML Pascal a snap. Unfortunately, if you did this there would be little room left for any program! After you pass 16K of text in MacPascal things get very slow, eventually it will die altogether. This is not my last word on this issue. Next time I'm using inline I'll bring it up again.

## MacPascal Books

There are now five books about learning Pascal using MacPascal. Some seem better than others. All have many interesting, short, Pascal examples. I don't know what to say. There are too many of them!

## MacPascal Books I Have Heard Of

**Pascal for the Macintosh** by Andrew Singer and Henry Legard, Addison Wesley

**Macintosh Pascal** by Lowell A. Carmony and Robert L. Holliday, Computer Science Press

**MacPascal Programming** by Drew Berentes, Tab Books

**The MacPascal Book** by Paul Goodman and Alan Zeldin, Brady Communications Company, Inc. A Prentice-Hall Publishing Company

**Macintosh Pascal** by Robert Moll and Rachel Folsom, with Mary Elting, Houghton-Mifflin

## Other Books with Pascal Examples

**Macintosh Revealed Volume One and Two**, by Stephen Chernicoff, Hayden Book Company

**One Flew Over the Quickdraws Nest** by Greg A. Lewis, Valuable Information Press  
(Also available from the MacTutor Mail Order Store)

## Pascal Compiler

**MacLanguage Series Pascal: \$99**  
TML Systems  
582 North Wickham Rd., Suite 93  
Melbourne Fla. 32935  
(305) 242-1873

(Also available from the MacTutor Mail Order Store)

## TML Pascal Program for ShowRes Desk Accessory

program ShowResource; { file ShowRes.pas }

```
{
ShowRes is a desk accessory that displays
the resources that are on the heap.
Written 9/85 by Alan Wootton. Compile
with TML Pascal using make DA option.
Resource compile ShowRes.R to add menu and
dialog stuff and to convert CODE to DRV.
}
```

```
{ $I=Include these interface files }
(*$I MemTypes.ipas *)
(*$I QuickDraw.ipas *)
(*$I OSIntf.ipas *)
(*$I ToolIntf.ipas *)
```

```
(*$A+ Write source code to .ASM file *)
(*$B+ set bundle bit*)
```

(\*\$T ???? type and creator \*)

{ The values of the ParamBlock CScCode field will have the following values. These constants are not found in any of the Macintosh interface files. }

```
CONST accEvent = 64;
 accRun = 65;
 accCursor = 66;
 accMenu = 67;
 accUndo = 68;
 accCut = 70;
 accCopy = 71;
 accPaste = 72;
 accClear = 73;
```

## TYPE

```
Lptr = ^longint;
chBlock= packed array [0..15,0..15] of char;
chBlockPtr=^chBlock;
```

{ This is our data. A handle to it will be stored in the dctlStorage field of the DCtlEntry record }

```
mainDataP = ^maindata;
mainDataH = ^mainDataP;
mainData = record { our personal Data }
 TheType : ResType; { type of subject }
 TheRes : Handle; { to subject }
 TheID : integer; { ID of subject resource }
 ViewR : Rect; { lower portion of box }
 InfoR : Rect; { upper left }
 MaxR : Rect; { giant Rect }
 click : point; { where a mdown was }
end;
```

{ All drivers have one of these. It is used to store system information about the state of the driver. It will also be passed to us on all calls from system. Note that this record IS defined in the interface files above (as DCtlEntry) so it is NOT strictly necessary to redefine it here. I am doing it because I wish to use dctlStorage^^ to refer to MainData without coercing types. }

```
DeviceControlRec = record
 dCtlDriver : Handle;
 DctlFlags : integer;
 dctlQueue : integer;
 DctlQHead : Lptr;
 DctlQtail : Lptr;
 DctlPosition : longint;
 DctlStorage : maindataH;
 dCtlRefNum : integer;
 dCtlCurTicks : longint;
 dCtlWindow : GrafPtr;
 dCtlDelay : integer;
 dCtlEmask : integer;
 dCtlMenu : integer;
end;
```

```
{ var variables for main program; not used }
```

```
{ -----
--- This first section is some data drawing procs.
----- }
```

```
{ Draw one hex character in current port
{ at current pen position. }
```

```
procedure PlotHex (c : char);
begin
 if ord(c) < 10 then
 DrawChar(chr(ord(c) + ord('0')))
 else
 DrawChar(chr(ord(c) + ord('A') - 10));
end;
```

```
{ Draw one Hex byte. }
```

```
procedure PlotByte (c : char);
begin
 PlotHex(chr(ord(c) div 16));
 PlotHex(chr(ord(c) mod 16));
end;
```

```
{ Draw 256 Hex bytes, and 256 ASCII chars. }
```

```
procedure PlotHexBlock (chptr: chBlockPtr);
var
 i, j : integer;
 xy : point;
 ch : char;
begin
 TextMode(srcOr);
 GetPen(xy);
 for i := 0 to 15 do
 for j := 0 to 15 do
 begin
 ch:=chptr^[i,j];
 MoveTo(xy.h+j*16, xy.v+i*12);
 PlotByte(ch);
 MoveTo(xy.h+j*8+256, xy.v+i*12);
 DrawChar(ch);
 end;
 TextMode(srcCopy);
 end;
```

```
{ This recursive little beast draws longints without
leading zeros. But it won't draw zero. }
```

```
procedure DrawLong (i : longint);
begin
 if i < 0 then
 begin
 i := -i;
 DrawChar('-');
 end;
 if i > 0 then
 begin
 DrawLong(i div 10);
```

```
 DrawChar(chr((i mod 10) + ord('0')));
 end;
end;
```

```
{ Draw an integer, or '0' if none. }
```

```
procedure DrawInteger (i : integer);
begin
 if i = 0 then
 DrawChar('0')
 else
 DrawLong(i);
end;
```

```
{ This works just like DrawLong above
but draws the number in binary }
```

```
procedure DrawBinary (i : integer);
begin
 if (i < 0) and (i > -1) then
 begin
 DrawBinary(i div 2);
 DrawChar(chr((i mod 2) + ord('0')));
 end;
end;
```

```
{ convert a ResType into a str255 }
```

```
procedure TypetoStr (var thetype : Restype;
var str : str255);
begin
 str := '1234'; { so str has right length }
 BlockMove(@thetype, @str[1], 4);
end;
```

```
{ -----
--- Here are the procs that draw data into the
--- window and calculate where in the data a
--- click occurred.
----- }
```

```
{ Given a click in the list of IDs,
calculate which Resource it was. }
```

```
procedure ClickID (var MainVars : MainData);
var
 i : integer;
begin
 with MainVars do
 begin
 { first we find the x part (mod 4) then we
 add the y part (* 4), then we have an index }
 i := ((click.h - ViewR.left) div 72) mod 4;
 i := i + (((click.v - ViewR.top) div 12) * 4);
 if i > CountResources(TheType) then
 i := 0;
 if i > 0 then
 begin
 SetResLoad(False);
 TheRes := GetIndResource(TheType,i);
```

```

 SetResLoad(true);
 end
else
 TheRes := nil;
end;
end;

{ Given a click in the list of types, calculate which type it was. }

procedure ClickType (var MainVars : MainData);
var
 i : integer;
begin
 with MainVars do
 begin
 { this works the same as above except there
 are 8 items per line, not 4 }
 i := ((click.h - viewR.left) div 36) mod 8;
 i := i + (((click.v - viewR.top) div 12) * 8);
 if i > countTypes then
 i := 0;
 if i > 0 then
 GetIndType(TheType,i)
 else
 TheType:=ResType(pointer(0));
 end;
 end;
end;

{ Draw list of types in ViewR. }

procedure DrawTypes (var MainVars : MainData);
var
 i : integer;
 tmpType : ResType;
 str : str255;
begin
 with MainVars do
 begin
 moveto(infoR.left, infoR.top + 12);
 DrawString('Click on one of the types below');
 for i:=1 to countTypes do
 begin
 moveto(viewR.left, viewR.top + 12);
 move((i mod 8) * 36, (i div 8) * 12);
 GetIndType(tmpType,i) ;
 TypeToStr(tmpType, Str);
 DrawString(str);
 end;
 end;
 end;
end;

{ Draw list of ID's in ViewR. }

procedure DrawIDs (var MainVars : MainData);
var
 tmpID, i : integer;
 tmpName, str : str255;
 H : Handle;
 tempType : ResType;
begin
 with MainVars do

```

```

begin
 moveto(infoR.left, infoR.top + 12);
 TypeToStr(TheType, str);
 DrawString('These are the resources of type ');
 DrawString(str);

 for i:= 1 to CountResources(TheType) do
 begin
 moveto(viewR.left, viewR.top + 12);
 move((i mod 4) * 72, (i div 4) * 12);

 SetResLoad(False);
 H:=GetIndResource(TheType,i);
 SetResLoad(true);
 GetResInfo(H,tmpID,tempType,tmpName);

 DrawChar(' ');
 DrawInteger(tmpID);
 DrawChar(':');
 DrawString(tmpName);
 end;
 end;

{ Draw Info about the Resource. }

procedure DrawResInfo (var MainVars : MainData);
var
 tmpID : integer;
 tmpName, str : str255;
 H : Handle;
 tempType : ResType;
begin
 with MainVars do
 begin
 if TheRes <> nil then
 begin
 tmpName := '';
 GetResInfo
 (TheRes,tmpID,TempType,tmpName);
 TypeToStr(tempType, str);

 moveto(infoR.left, infoR.top +8);
 DrawString('TYPE:');
 DrawString(str);
 DrawString(' Name:');
 DrawString(tmpName);

 moveto(infoR.left, infoR.top + 18);
 DrawString('ID:');
 DrawInteger(tmpID);
 DrawString(' attributes:');
 DrawBinary(GetResAttrs(TheRes));
 DrawString(' Size:');
 DrawLong(SizeResource(TheRes));

 MoveTo(infoR.left, infoR.top+ 28);
 DrawString(' Handle is:');
 DrawLong(ord(TheRes));
 DrawString(' belongs to map#');
 DrawInteger(HomeResFile(TheRes));
 end;
 end;
 end;
end;

```

```

 MoveTo(viewR.left, viewR.top + 12);
 if TheRes^ <> nil then
 PlotHexBlock(pointer(ord(TheRes^)));
 end;
 end;
end;

{ =====
ooo Following are procs to service the window.
ooo Windows need updating, and activation, etc
===== }

{ a window activation function. We take this }
{ opportunity to put our menu in the menu bar }

procedure activate (var device : deviceControlRec);
var
 menH : menuHandle;
begin
 with device do
 begin
 menH:= GetMenu(dctlMenu);

 { the Font/DA mover does not correct the
 ID of menus when it rennumbers them.
 So, we must do it here }
 menH^.menuID:=dctlMenu;

 InsertMenu(MenH,0);
 DrawMenuBar;
 end;
 end;

{ A window deactivation function. We must remove }
{ our menu from the bar at this time }

procedure deactivate(var device:deviceControlRec);
begin
 with device do
 begin
 deleteMenu(dctlMenu);
 DrawMenuBar;
 end;
 end;

{ Redraw the window. Draw type list, or ID list, or
 resource information depending upon what is set. }

procedure updateself (var device:deviceControlRec);
var
 max : integer;
begin
 with device do
 begin
 beginupdate(dctlWindow);
 drawdialog(dctlWindow);
 with dctlStorage^^ do { with MainData }
 begin
 if Longint(TheType)=0 then

```

```

 DrawTypes(dctlStorage^^)
 else
 if TheRes = nil then
 DrawIDs(dctlStorage^^)
 else
 DrawResInfo(dctlStorage^^);
 end;
 endupdate(dctlWindow);
 end;
 end;

{ =====
ooo Finally we have the three main procs.
ooo Open, Ctl, and Close. All calls from the
ooo 'outside' will be handled by one of these three.
ooo Note that Ctl has the event case split out
ooo for easier readability.
===== }

{ Open:The first call from the system. Make a
 window and setup our private storage.
 Note that we could get an open call even after we
 are already open. }

procedure open (var device : deviceControlRec;
 var block : ParamBlockRec);
var
 Wpk : WindowPeek;
 Dtyp, ID : integer;
 Dhan : Handle;
 tmptr : ptr;
 dummy:MainData;
begin
 ID := $C000 - 32 * (1 + device.dctlRefNum);
 { use this ID for all resource access }
 device.dctlMenu:=ID;

 with device do
 if dctlWindow = nil then
 begin
 dctlStorage := pointer(NewHandle(SizeOf(dummy)));
 { This is a handle to our private data }

 { It is good voodoo to keep window records }
 { off of the bottom of the applications heap }
 tmptr:=NewPtr($1000); { 4K on bottom of heap }

 dctlWindow:=GetNewDialog(ID,nil,nil);

 { Must set windowKind field. Never forget this !}
 wpk := pointer(ord(dctlWindow));
 wpk^.windowKind := dctlRefNum;

 setport(dctlWindow);
 textMode(srcCopy);
 textfont(3);
 textsize(9);

 disposptr(tmptr);
 { leaving 4K hole under window record}
 end;
 end;
 end;
end;

```

```

Hlock(Handle(DctlStorage));
with DctlStorage^^ do
begin
 { initialize our storage }
 GetDitem(dctlWindow,2,Dtyp,Dhan,viewR);
 GetDitem(dctlWindow,3,Dtyp,Dhan,InfoR);
 TheType:=ResType(pointer(0));
 TheRes := nil;
 SetRect(MaxR, -999, -999, 999, 999);
end;{ of with storage }
Hunlock(Handle(DctlStorage));
end;{ of with }
end;{ of open }

```

{ This is the end and there is no way to say no.  
The last call we will ever get.  
We must clean up our act completely. }

```

procedure close (var device : deviceControlRec;
 var block : ParamBlockRec);
begin
 deactivate(device);{ remove menu }
 with device do
 begin
 disposHandle(Handle(dctlStorage));{ kill data }
 disposDialog(dctlWindow);{ erase window }
 dctlWindow := nil;
 end;{ of with }
end;{ of close }

```

{ We are here because there has been an event  
concerning our window.  
Block.csParam[0] and [1] point at the event record }

```

procedure Event (var device : deviceControlRec;
 var block : ParamBlockRec);
var
 item, part : integer;
 tmpDptr:DialogPtr;
 evP:^EventRecord;
begin
 blockMove(@block.csParam[0],@evP,4);
 { above is the best way I could think of to
 convert 2 integers into a pointer }

 with evP^ do { with eventRecord }
 with device.dctlStorage^^ do { with MainData }
 begin
 case what of
 MouseDown :{1}
 begin
 click:= where;
 globalToLocal(click);
 if dialogselect(evP^,tmpDptr,item) then
 { normally one would compare tmpDptr
 with ones own window, but a DA won't
 get an event unless it's in his window }
 begin
 case item of
 1 : { close }
 if TheRes = nil then
 TheType := ResType(pointer(0))

```

```

else
 TheRes := nil;
2 : {viewRect }
begin
 if Longint(TheType) = 0 then
 ClickType(device.dctlStorage^^)
 else
 if TheRes = nil then
 ClickID(device.dctlStorage^^);
 end;
 otherwise
 ;
 end; { of item case }
 InvalRect(MaxR); { force update }
 end; { if DialogSelect }
end; { Mdown case }
UpdateEvt : {6}
 updateself(device);
ActivateEvt : {8}
 if odd(modifiers) then
 activate(device)
 else
 deactivate(device);
 otherwise
 ;{ ignore all other events }
 end;{ case of what }
end;{ with eventRecord }
end;{ procedure event }

```

{ Here is the main entry point for system calls.  
The parameter block tells us what the nature of  
this call is. Note that in a more complex program  
the menu case could be made into a separate proc  
like the event case. }

```

procedure ctl (var device : deviceControlRec;
 var block : ParamBlockRec);
var
 poi : point;
 cHan : cursHandle;
 wpnt : GrafPtr;
 item : integer;
begin
 setport(device.dctlWindow);
 Hlock(Handle(device.dctlStorage));
 with device.dctlStorage^^ do
 with block do
 begin
 case csCode of
 accEvent:
 Event(device, block);
 accCursor:
 begin
 GetMouse(poi);
 if ptInRect(poi, ViewR) then
 begin
 cHan:=GetCursor(CrossCursor);
 SetCursor(cHan^^);
 end
 end
 else

```

```

 initcursor;
 end;
accMenu:
 begin { case out menu item number }
 case csParam[1] of
 1 : { about... }
 begin
 Wpnt:=GetNewDialog(
 device.dctlMenu+1,nil, pointer(-1));
 ModalDialog(nil,item);
 DisposDialog(wpnt);
 end;
 3 : { LoadResource }
 if TheRes <> nil then
 LoadResource(TheRes)
 else
 sysbeep(10);
 4 : { ReleaseResource }
 if TheRes^ <> nil then
 begin
 ReleaseResource(TheRes);
 TheRes := nil;
 end
 else
 sysbeep(10);
 otherwise
 sysbeep(10);
 end;{ of case of menu }
 InvalRect(MaxR);{ force update }
 HiliteMenu(0);
 end;{ menu case of code }
 otherwise
 ;
 end;{ case of code }
 end;{ of with block }
 Hunlock(Handle(device.dctlStorage));
end;{ of control }

```

```

{
ooo Observe the poor main procedure. Since we
ooo will link with another module that will call
ooo Open, Ctl,and Close directly; there will
ooo never be any use for a main proc.
ooo Furthermore, there can be no global data for
ooo a main proc to use.
.....

```

```

BEGIN
(* No main program allowed *)
END.

```

```

;;=====
;; file ShowRes.R
;;
;; Feeding this to Rmaker is the last
;; step when compiling ShowRes.
;;
;; The CODE 1 resource is read from
;; ShowResL, the link output
;;
;; A Font/DA Mover data file is made.
;;=====

```

```

;;= All Resources must be between
;;= -15872 and -15841 inclusive if they
;;= are to travel with DRVr number 16

```

```

;;=====
ShowRes.acc;; destination file name
DFILDMOV;; type and creator
;;=====

```

```

type DRVr = PROC
ShowRes,16;; name, DRVr number 16
ShowRes

```

```

;;=====
;;= This DLOG is the main window.
;;= The StatText items are used only
;;= for their rectangles.
;;=====

```

```

type DLOG
ShRsDLOG,-15872;;id owned by #16
Show Resource;;;window title
53 55 260 469 ;;global bounds
Visible GoAway
4 ;;wind. type 4
0 ;;refcon 0
-15872 ;;id of DITL

```

```

type DITL
ShRsDITL,-15872
3

```

```

BtnItem Enabled
5 320 25 411
Close

```

```

StatText Enabled
25 5 206 410
;;; View, clicks taken in here

```

```

StatText Enabled
0 5 26 225
;;; Info, title to data drawn here

```

```

;;=====
;;= This is the menu used by the DA.
;;=====

```

```

type MENU
ShRsMENU,-15872
ShowRes;; menu title
about..;; menu1
(-;; dotted line
LoadResource;; menu3
ReleaseResource;;menu4

```

```

;;=====
;;= This second DLOG is for the
;;= 'about..' menu item. The button
;;= and icons do nothing. They are
;;= only for decoration.
;;=====

```

```

type DLOG
ShRsDLOG2,-15871
About ShowRes;; title

```

60 50 300 455;; size  
Visible NoGoAway  
4  
0  
-15871

type DITL  
ShRsDITL2,-15871  
10

StatText enabled  
5 5 91 400  
ShowRes - an accessory to view ++  
resources on the heap. Created ++  
by Alan Wootton 9/85. ++  
Written using Macintosh Pascal ++  
and MacLanguage Pascal ++  
from TML systems, Melbourne Fla. ++  
Published in MacTutor, The Macintosh ++  
Programming Journal.;;

BtnItem enabled  
95 5 102 395  
{|||||||};;;used as separator

StatText enabled  
105 5 191 400  
To select Resources click on the ++  
list displayed. Use the button to ++  
return to a previous list. ++  
Resources loaded will show the ++  
first 256 bytes. ++  
Load resources with "LoadResource" ++  
menu item. Use "ReleaseResource"++  
carefully!!!;;

IconItem Disabled;; Icons along bottom  
195 20 227 52 ;; for fun.

0  
IconItem Disabled  
195 75 227 107

1  
IconItem Disabled  
195 130 227 162

2  
IconItem Disabled  
195 185 227 217

1  
IconItem Disabled  
195 240 227 272

2  
IconItem Disabled  
195 295 227 327

1  
IconItem Disabled  
195 350 227 382

0

;;~~~~~  
;;= end of file ShowRes.R  
;;~~~~~





# Pascal Procedures

## Christmas Graphics



Alan Wootton  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 13

Last month I promised to complete my discussion on desk accessories. I still plan to do that, but this month is Christmas, so we will take it easy and do some more lighthearted things. We will take an overview of the types of programming one might do on the Macintosh. I have two short programs demonstrating how to write code in resources, and a short program that draws a Christmas tree.

### Resources As Code

Unless you are writing code for a dedicated appliance controller or something similar, the programs you write will be started by another program, usually the operating system. Your program, when finished, will return control to that 'other' program. In the case of a Macintosh Application, the program is started by another Application (usually the Finder) using the Toolbox procedure Launch (see Segment Loader chapter of Inside Mac). When the program is finished, it returns to the Finder by calling ExitToShell (or Launch!), or by just reaching the end of the main procedure, in which case ExitToShell is called for you. If you have been following this column, you will know that there are other kinds of programs besides just Applications. Desk Accessories are called repeatedly by the toolbox when the resident Application calls SystemTask. In addition to these two, there are many other ways to get your piece of code executed.

One of the features of native code compilers (for the 68000) is that the code they produce is position independent. This means that it does not matter what address the program occupies: it always runs properly. So potentially any portion of memory could be loaded with code and run. In the case of an application, the Segment Loader handles this chore. In the case of DAs, the Device Manager loads resources of type DRVR and passes control to the code within. The Window Manager, the Menu Manager, the Control Manager, and others will load and run pieces of code. There is no reason why we cannot do it, too.

The compilers normal output (actually produced by the linker) is a single piece of code that is stored in the resource CODE #1. The structure of this code is illustrated in figure 1.

Note that the code is produced in the same order as the procedures occur. A jump is used to transfer control to the main procedure. Further, note that the main procedure does not begin and end like the others. It does not return control to the caller at all. Rather, it exits to the finder. The link and unlink are used to reserve space on the stack for the vars declared for that procedure.

| Pascal Source                              | 68000codeproduced                                                              |
|--------------------------------------------|--------------------------------------------------------------------------------|
| Program Ex;                                | dc.w 0,1<br>jmp ex                                                             |
| procedure proc1;<br>begin<br>...<br>end;   | proc1 link A6,#...<br>...<br>unlk A6<br>rts                                    |
| procedure proc2;<br>begin<br>...<br>end;   | proc2 link A6,#...<br>...<br>unlk A6<br>rts                                    |
| begin { main }<br>proc2;<br>proc1;<br>end. | ex move.l sp,A6<br>bsr proc2<br>bsr proc1<br>_ExitToShell                      |
|                                            | lib1 move.l (sp)+,D0<br>_Trap<br>move.l (sp)+,A1<br>move.l a0,(sp)<br>jmp (a1) |

Figure 1. Illustration of the structure of Pascal produced native 68000 code.

An important thing to notice is the existence of phantom procedures past the end of the program. These are known as library routines and have the purpose of providing functions that are needed by the program. String functions and Operating System traps are examples. You can even make your own library procedures and, by declaring them, cause the linker to include that code.

There is a problem with the MDS linker. It is not as efficient as it should be. When you refer to a .rel file in a link directive file (link directive files are produced automatically by TML Pascal, although you may make your own), the whole .rel file is linked onto the end of your code, whether it is used or not. If you need more control, you can make your own library files that only have those routines actually needed in them (requiring much work in assembly language). Another way is to get the Optimizing Linker and Librarian from Consulair Corp. (Portola Valley, CA). Consulair normally sells a C compiler, but since that compiler is MDS compatible, you may use their products in conjunction with

the TML Pascal Compiler. TML may be able to sell you that linker also.

There is another oddity about the main procedure that is not shown in the diagram. All variables declared in the main procedure are accessed in a special way. These variables are not created temporarily like in the other procedures. The variables for the main procedure are created by the segment loader above the beginning of the stack and the register A5 is *reserved* for the sole purpose of accessing those variables. This means that, except for application code, *procedures in resources absolutely must not access any global variables.*

This was mentioned in regard to DAs last month and is applicable here, too.

Before we lose our way in all this technical mush, let us remember that our goal is to use a resource as a procedure. In order to do this we have one final hurdle to overcome.

The most general method will be to use the beginning of the resource as the beginning of the program. A glance at figure 1 shows that this does not work on the program in its CODE 1 form. Since we will have to use the Rmaker to convert from type CODE to another type, (here we use PROC) we could use Rmaker to convert from the program form to the procedure form. Rmaker does not make this an easy task. There is a Rmaker type that will trim off the first two words (created by the segment loader) of the CODE 1 resource (refer to your Rmaker documentation). Unfortunately, we need to also avoid the jump to the main procedure. If we tell the compiler to put our procedure into a CODE 2 resource then there is no jump, but Rmaker will not trim any other than CODE 1. After much thought I came upon what seems to be the best solution. Use the GNRL type directive to place the word \$6008 at the front of the destination resource and then copy the CODE 1 resource after that. \$6008 is the 68000 code to branch over the next 8 bytes. This effectively skips the two segment loader bytes and the jump. For CODE types other than number 1 there will not be a jump, so use \$6004 to skip only the first two words. See the Rmaker input file below for an example.

### Prompt\_For\_String

The procedure I present to illustrate the use of procedures in resources is a compiled version of the Dialog box example presented here in July (Vol. 1 No. 8). Its purpose is to open a dialog box, request a string from the user, and then go away, returning the string to the calling program. To compile this first compile Pr\_for\_Str.pas, link Pr\_for\_Str.link (this file is created by the compiler), and then use Rmaker with Pr\_for\_String.R. The resulting file is Pr\_for\_Str.PROC which contains the resources PROC 567, which is our procedure, and also the DLOG and DITL for the dialog box.

The most interesting part of the example is the third part, the program Pr\_for\_Str.Mpas. This is a short (except for the

declarations) MacPascal program that shows how you could use an external and compiled procedure from within MacPascal.

### Inline Again

All the action in Pr\_for\_Str.Mpas (below) takes place in the main procedure. For readability of those eleven lines, I have declared the Toolbox routines as procedures, rather than just using inlines in the code. This is the simplest use of inline: once you get the definitions correct it is difficult to call the Toolbox traps incorrectly. Following the example below it is easy to type in Toolbox routines using Inside Mac as a guide.

The exceptions are register based traps. For these I use Generic (see Advanced Mac'ing in MacTutor Vol.1 No. 5). When Inside Mac says (using Hlock as an example):

```
PROCEDURE Hlock(h:handle);
 On_entry A0: h (handle)
 On_exit D0: result code (integer)
```

You set regs.a[0] to the handle and call Generic with \$A029 (look up the trap number in a cross reference). The record regs is read by Generic and the trap is called. Note that later you can check loword(regs.d[0]) for an error.

This is not the end of the the ugliness. In order to call an external procedure it is necessary to abuse inline. Instead of a trap number we use the word \$4E75 which will transfer execution to the address corresponding to the last argument to inline. The last argument is @jsr[0] which is the address of an array of 4 words which has been set to a short routine to shuffle some registers and then call the address that is next to last in the arguments. In this case it is the address of the resource PROC 567. For more info on jsr see column in MacTutor Vol.1 No.9. The printing example uses jsr and the 68000 code is given.

Once you become comfortable with the inline kludges required, you can make arbitrarily complex MacPascal programs. Simply work on your program until it becomes too large (or too slow), then compile those procedures that are in a relatively finished state. When the compiled procs are replaced by the code to call them externally, your program will then be much shorter, and you can add to it until it is time to repeat the cycle. Eventually, all that is left is a core with the bulk of the program compiled. At that point you move the last of the program to the compiler and you have a completed program!

Skip over the Pr\_for\_Str stuff (3 files) now and we will do something much more fun.

### TML Pascal Code

```
Program Pr_For_Str;{ file Pr_For_Str.pas }
{ by Alan Wootton 10/85 }
```

```

{ written for TML Pascal }

{ $I means include these interface files }
($I MemTypes.ipas *)
($I QuickDraw.ipas *)
($I OSIntf.ipas *)
($I ToolIntf.ipas *)

{ We will convert this code with Rmaker in
such a way as to cause execution to begin
with the FIRST PROCEDURE, and not in the
main procedure. }

{ This procedure expects the resource
DLOG 12345 to be available. It opens a
dialog box and returns a string in result.
If no string is input by the user then
the string "" is returned. }

{ Execution begins here }
Procedure Prompt_for_Str(var Prompt:str255;
 var Sample:str255;
 var Result:str255);

const
 OKbutton =1;{ items in DLOG box }
 CANCELbutton =2;
 RESULTtext =3;
 PROMPTtext =4;
var
 DlogPtr : DialogPtr;
 TempHand: handle;
 itype, itemHit : integer;
 R : rect;
 itemH : Handle;
begin
 TempHand:=GetResource('DLOG',12345);
 if TempHand<>nil then
 begin
 DlogPtr:=GetNewDialog(12345,nil,pointer(-1));

 GetDItem(DlogPtr,PROMPTtext,itype,itemH,R);
 if length(Prompt)>0 then
 SetText(itemH, Prompt);

 GetDItem(DlogPtr,RESULTtext,itype,itemH,R);
 if length(Sample)>0 then
 SetText(itemH, Sample);
 { Note that itemH is now handle to result }
 { text item and will be used later. }

 ModalDialog(nil, itemHit);

 if itemHit=CANCELbutton then
 result := ""
 else
 GetIText(itemH, result);

 DisposDialog(DlogPtr);
 end;
 end;{ of procedure }
begin{ main }

```

```

{ !!!! main not used !!!! }
end.

```

### Rmaker code

```

=====
;; file Pr_For_Str.R
;;
;; Feeding this to Rmaker is the last
;; step when compiling Pr_For_Str
;;
;; The CODE 1 resource is read from
;; Pr_For_Str, the link output, and
;; is written to the resource PROC 567
;; in Pr_For_Str.PROC
;;
;; A branch is added to the front of
;; the code to skip the segment header
;; (4 bytes), and in this case, to also
;; skip the instruction to jump to the
;; main procedure (4 bytes, for 8 total).
;; Use 6004 for bra.s *+4.
=====
Pr_For_Str.PROC;;; destination file name
????????;; type and creator
=====

type PROC = GNRL
Prompt_For_Str,567
.H
6008;; bra.s *+8
.R
Pr_For_Str CODE 1

**
** Definition of a Dialog Manager **
** window. **

;; global coordinates !!

type DLOG
box,12345
;;notitle
96 128 148 384 ;; top left bottom right
visible goaway
1 ;; window type = dBoxProc
0 ;; refcon
12345 ;; ID of DITL associated
 ;; with this DLOG

**
** Next is a list of 'items' to go **
** in the window. **

type DITL ;; see Dialog Manager
items,12345
4 ;; four items

Button
4 120 24 180 ;; local coordinates !!
OK;;

```

```

Button
4 188 24 248
Cancel;;

EditText Disabled
32 8 48 248
::::::::::::::::::text added later

StaticText Disabled
4 8 20 120
Type a String;; prompt, modified later

```

```

;;=====
;;~ end of file Pr_For_Str.R
;;=====

```

### MacPascal Code

```

program Pr_For_Str_Test;{ by Alan Wootton 10/85 }
{ This program exercises an external procedure }
{ that presents a dialog box requesting the user }
{ to input a string. }
type
 ptr = ^char;
 handle = ^ptr;
 ResType = longint;
var
 ResRefNum : integer;{ resource file ref num }
 str : str255;
 regs : record { for generic }
 A : array[0..4] of longint;
 D : array[0..7] of longint;
 end;
{-----}
{--Toolbox interface routines we will be using-----}
{-----}
function OpenResFile (filename : str255) : integer;
begin
 OpenResFile := WinlineF($A997, @filename);
end;
procedure CloseResFile (refNum : integer);
begin
 inlineP($A99A, refnum);
end;
function HomeResFile (TheResource : Handle) : integer;
begin
 HomeResFile := WinlineF($A9A4, TheResource);
end;
function GetResource (TheType : ResType;
 TheID : integer) : Handle;
begin
 GetResource := pointer(LinlineF($A9A0, TheType, TheID));
end;
procedure DetachResource (TheResource : Handle);
begin
 inlineP($A992, TheResource);
end;
{ The Hlock that is predefined does not work!!! }
procedure Hlock (H : Handle);
begin

```

```

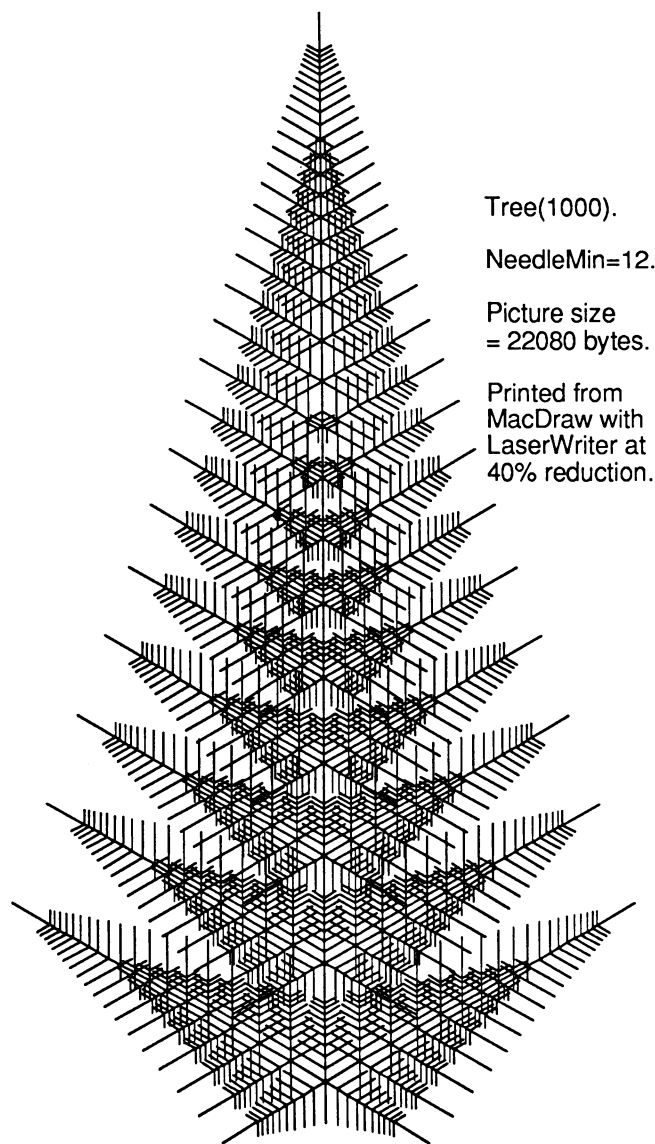
 regs.a[0] := ord(h);
 Generic($A029, regs);
end;
{ convert a Str255 to ResType }
function StrToType (str : str255) : ResType;
var
 TheType : ResType;
begin
 BlockMove(@str[1], @TheType, 4);
 StrToType := TheType;
end;
{--end of interface routines-----}
{-----}
{--Routine to call PROC 567 resource -----}
{-----}
procedure Prompt_for_String (Pr : str255;
 Sa : str255;
 var Re : str255);
var
 Hand : handle;
 jsr : array[0..3] of integer;
begin
 stuffHex(@jsr, '5488225F2F084ED1');
 { code to jsr to top of stack }
 Hand := GetResource(StrToType ('PROC'), 567);
 if Hand <> nil then
 begin
 Hlock(Hand);
 inlineP($4E75, @Pr, @Sa, @Re, Hand^, @jsr);
 { $4E75 is rts to code in @jsr which calls Hand^ }
 { normally you might unlock Hand now }
 end
 else
 writeln('PROC 567 not found');
 end;
begin { main, test prompt for string }
 ResRefNum := OpenResFile('Pr_For_Str.PROC');
 if ResRefNum > 0 then
 begin
 Prompt_for_String('Str Please', 'example str', str);
 writeln('The str returned is ', str);
 CloseResFile(ResRefNum)
 end
 else
 writeln('OpenResFile failed');
end.

```

### Christmas Graphics

For those who like short MacPascal programs that make intricate drawings I present the program X\_Tree (below) that makes the picture-of-many-needles (above). The way this works is that it loops, while drawing branches (needles) proportional to the remaining length, until the remaining length is short. To draw the branches the same routine is called again (an example of recursion), so that the branches

look like smaller versions of the whole tree. This would be a simple program except that in order to project a line of a particular length (Length), in a particular direction (Direction) one must use the trigonometric functions Sin and Cos. These functions, when multiplied by a length, give the horizontal (Cos) and vertical (Sin) component of a line in the given direction. It wouldn't be so bad, but the direction given to Sin and Cos is not in degrees! It is in radians. Radians are a mathematical unit for angles used by SANE and scientists everywhere. To use radians, note that  $180^\circ$  is the same as  $\pi$  radians (see how the variable pi is set to the value of  $\pi$  below). This means that  $60^\circ$  (or  $180^\circ/3$ ) is the same as  $\pi/3$ .



Output of the X\_tree Program

To make the drawing above, I pasted the DrawSomething procedure into the program Pict\_to\_Clip (MacTutor, Vol.1 No.11) and ran it. I then quit MacPascal and started MacDraw and did a paste. After that I added the text next to it and cut everything onto the clipboard. I then pasted the result into MacWrite for this article. [The Pict\_to\_Clip utility referred to above is a marvelous little program that writes a user-defined function to a pict resource and moves the pict resource into the clipboard, where it can be pasted into other applications that support laser printing, like MacDraw. In this way, it makes the Macintosh into a plotter! It's available on our source code disks or as a back issue (October 1985) through the MacTutor mail order store. -Ed.]

```

program X_Tree;
uses
 SANE;

procedure DrawSomething;
const
 NeedleMin = 5; { cutoff size for Needles }
var
 pi, Direction, X, Y : extended;

{ The variables above are global to the proc "Tree". }
{ X, and Y, are the pen position in floating point form. }
{ Direction is an angle pointing in the direction the }
{ "tree" is. Direction is in radians, ie. -pi/2 is up, }
{ pi/2 is down, 0 is to the right, pi is to the left. }

procedure Tree (Length : extended);
{ Given a length and a direction, this proc will }
{ draw a line of the given length and, size permitting, }
{ will draw a series of "subtrees", or "needles", of }
{ decreasing size alongside the "tree" line. }
var
 OldDir, Needle, PrevX, PrevY : extended;
begin
 PrevX := X;
 PrevY := Y; { Save direction and position. }
 OldDir := Direction;
 Needle := Length / 3; { Length of first "needle". }

 while Length > 1 do
 begin { Subdivide "tree" }

 If Needle > NeedleMin then
 begin { Draw left, then right, needle. }
 Direction := OldDir - pi / 3; { 60 degrees }
 Tree(Needle);
 Direction := OldDir + pi / 3;
 Tree(Needle);
 end
 else { else make line remaining length }
 Needle := Length * 3;

 { Draw portion of tree between successive needles }
 Direction := OldDir;
 MoveTo(num2integer(X), num2integer(Y));
 end
 end
end

```

```

X := X + (Cos(Direction) * Needle / 3);
Y := Y + (Sin(Direction) * Needle / 3);
LineTo(num2integer(X), num2integer(Y));

Length := Length - Needle / 3; { shorten length }
Needle := Needle * (1 - 1 / 9); { shorten needle }
end;

X := PrevX;
Y := PrevY; { restore position and direction }
Direction := OldDir;
end;

begin { procedure DrawSomething }
pi := arctan(1) * 4; { 3.14159... }
Direction := -pi / 2; { -90 degrees = up }
X := 200;
Y := 240; { tree base at 200,240 }
Tree(200); { 200 = size of tree }
end;

begin { main program }
ShowDrawing;
DrawSomething;
end.

```

### More Resources As Code

A more technical example of writing code for resources is now presented for advanced programmers. The VBLEExample (below) is an INIT resource that installs a Vertical Retrace routine at system startup.

Vertical Retrace routines are short pieces of code that are executed periodically by the system. They are not for general use, however, since they are executed during an interrupt. This means that you *cannot use any Toolbox traps that use the memory manager*. This includes Quickdraw. Bob Denny (in C Workshop, MacTutor Vol.1 No.9) gives a good description of the Vertical Retrace Manager, so I won't do it here. All the example does is increment the first longint in the screen buffer. This makes a tiny binary counter in the upper left of the screen.

To put a task into the Vertical Retrace queue you must fill out a short record and pass it to Vinstall. Notice that I break all the rules and put this record in the same resource with the code! The procedure dummy is declared to make some unused space and GetGlobalData is called to get a pointer to our permanent storage record (remember, only the application can have permanent global variables).

Inlines again! The TML Pascal compiler has a type of inline that you can use (carefully). It is *not* the same as the MacPascal inlines. The syntax is a procedure declaration, followed immediately by "inline", and then by an integer. When the procedure is called the word is executed in the place of the normal jsr. In the program VBLEExample I create a procedure that will set register A0 and another that will invoke the trap \_Vinstall. By using these two it is possible to

alleviate the need for any libraries at all (no other Toolbox calls or procedures require library support in the example).

Bob Denny also gives a good description of how INIT resources work, so I won't repeat it. Debugging is another story. The code in INIT resources is called during startup and at that time it is just about impossible to use a debugger! This can make INIT resources very hard to trace. To make it easy I wrote the MacPascal program Run\_INIT (below). This program does to an INIT resource the same thing the system does at startup. It is also a good description of how the system treats INIT resources. Note that since no parameters are passed you can use Generic to call the external procedure instead of @jsr like it did in Pr\_for\_Str.Mpas. Otherwise these two have much in common.

We have seen the operation of two types of code resources, INIT and PROC. These are merely the tip of the iceberg. Perhaps later we'll try MDEF, WDEF, or CDEF (menu, window, and control definition functions) in addition to our normal projects involving CODE and DRVr (for applications and desks accessories).

Until next month, may the bugs bite you only in obvious places, and Merry Christmas!

```

program VBLEExample; { file VBLEExample.pas }
{ by Alan Wootton 10/85 }
{ written in TML Pascal }

```

```

{ $I=Include these interface files }
(*$I MemTypes.ipas *)
(*$I QuickDraw.ipas *)
(*$I OSIntf.ipas *)
(*$I ToolIntf.ipas *)

```

```

{ We will convert this code with Rmaker in
such a way as to cause execution to begin
with the FIRST PROCEDURE, and not in the
main procedure. }

```

### TYPE

```

GlobalDataP=^GlobalData;
GlobalData=record
 vblPart:VBLTask;
 count:longint;
end; { 18 bytes long }

```

```

{ var
 no global variables allowed }

```

```

{ the following four procs don't generate code now }
Procedure SetA0(a0:longint);inline $205F; { MOVE.l (SP)+,A0 }
Procedure Vinstall_Trap;inline $A033; { _Vinstall trap }
Function GetGlobalData : GlobalDataP;FORWARD;
Procedure VBLScreenTask;FORWARD;

```

```

{ execution begins here }

```

```
{ We install VBLScreenTask in the VBL queue and
 set up the body of the dummy procedure as our
 data record. }
```

```
Procedure InstallVBLTask;{ one time setup routine }
var
 cp : GlobalDataP;
begin
 cp := GetGlobalData;
 cp^.count:=0;
 with cp^.VBLPart do
 begin
 qType:=ord(vType);
 vblAddr:=@VBLScreenTask;
 vblCount:=1;
 vblPhase:=0;
 { This funky double step is my way of calling Vinstall
 without having to link with another file. Register A0
 is set and then the trap is called. }
 SetA0(ord(@cp^.VBLPart));
 Vinstall_Trap;
 end;
end;
```

```
Procedure Dummy;{ reserve some bytes in the code space }
begin
 Dummy; Dummy; Dummy; Dummy;{ 8 bytes }
 Dummy; Dummy; Dummy; Dummy;{ 8 bytes }
 Dummy;
end;
```

```
Function GetGlobalData { : GlobalDataP};
begin
 GetGlobalData := pointer(ord(@Dummy));
end;
```

```
{ Reset the VBLCount so we remain in queue and
 utilise $824 (SCRNBASE global) to find address
 of the screen and write the count there. }
```

```
Procedure VBLScreenTask;
var
 cp : GlobalDataP;
 ScreenP:^longint;
begin
 ScreenP:=pointer($824);
 ScreenP:=pointer(ScreenP^);
 cp:=GetGlobalData;
 with cp^ do
 begin
 count:=count+1;
 with VBLpart do
 begin
 VBLCount:=1;
 ScreenP^:=count;
 end;
 end;
 end;
end;{ vbltask }
```

```
begin{ main }
 { !!! main procedure not used !!! }
```

end.

```
=====
;; file VBLEExample.R
;;
;; Feeding this to Rmaker is the last
;; step when compiling VBLEExample.
;;
;; The CODE 1 resource is read from
;; VBLEExample, the link output, and
;; is written to the resource INIT 16
;; in VBLEExample.INIT
;;
;; A branch is added to the front of
;; the code to skip the segment header
;; (4 bytes), and in this case, to also
;; skip the instruction to jump to the
;; main procedure (4 bytes, for 8 total).
;; Use 6004 for bra.s "+4.
=====
VBLEExample.INIT;;; destination file name
????????;; type and creator
=====
```

```
type INIT = GNRL
VBLEExample,16 (80)
.H
6008;; bra.s "+8
.R
VBLEExample CODE 1
```

```
=====
;; end of file VBLEExample.R
=====
```

```
program Run_INIT;{ by Alan Wootton 10/85 }
{ This MacPascal program is for testing }
{ INIT resources. It loads the INIT }
{ resource number 16 from the file named below, }
{ and runs it just as the boot code would at system }
{ startup. The handle is writeln'd and you are given }
{ the opportunity to invoke a debugger, and set }
{ breakpoints, if desired. }
```

```
type
 ptr = ^char;
 handle = ^ptr;
 ResType = longint;
var
 ResRefNum : integer;{ resource file ref num }
 str : str255;
 hand : handle;{ handle to code }
 longP : ^longint;
 regs : record { for generic }
 A : array[0..4] of longint;
 D : array[0..7] of longint;
 end;
```

```
{-----}
{--Toolbox interface routines we will be using-----}
{ copy routines from Pr_For_Str_test (above) }
```

```

{-----}
{--Routine to call a 68000 proc in memory-----}
{--note that the handle is not locked,-----}
{--and no parameters are passed-----}
procedure RunHandle (Hand : handle);
begin
 regs.A[0] := ord(hand^);{ set A0 }
 Generic($4E90, regs);{ $4E90 = JSR (A0) }
end;

begin { main, program starts here }
 ShowText;
 ResRefNum := OpenResFile('VBLExample.INIT');
 if ResRefNum > 0 then
 begin
 hand := GetResource(StrToType('INIT'), 16);
 if hand <> nil then
 begin
 if HomeResfile(hand) = ResRefNum then
 begin

```

```

 writeln('handle is ', ord(hand));
 writeln('run resource ? (y/n)');
 readln(str);
 if str = 'y' then
 begin
 DetachResource(hand);
 RunHandle(hand);
 longP := hand^;
 longP^ := $4E714E71;{ nop nop }
 end;
 end
 else
 writeln(' resource from wrong file');
 end
 else
 writeln(' resource not loaded ');
 CloseResFile(ResRefNum);
 end
 else
 writeln('OpenResFile failed');
 end.

```





# BASIC SCHOOL

## Alphabet Soup



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 1

Welcome to Basic School! This article is dedicated to those of us who enjoy programming in Basic. Programmers of structured languages have put Basic down for several years because it is not considered a structured programming language and many implementations of Basic lack enhancements which give it power to perform advanced functions. Also it has been said many times that it is much harder to follow the program flow in Basic than in other languages like Pascal. Those who complain about the limitations of Basic are sometimes quite justified in their complaints. After all, basic "BASIC" isn't very powerful and not very flexible at all. Any enhancement at all is an improvement. This has been the case up until now. Now, the enhancements that we find in Macintosh Basic are approaching the power of the more acceptable high level languages such as Pascal. Exploring Macintosh Basic computing power is what this column is all about.

In the last few months we have seen the evolution of the programming tools needed to program the Mac on the Mac. In the coming issues, we will explore the enhancements included in the new MacBasic and in Microsoft Basic Version 2.0. As of the writing of this article, Microsoft says they intend to release Microsoft Basic Version 2.0 by the end of October or sometime in November. I'm told that it includes many enhancements, which in my opinion should have been included from the very first introduction of Microsoft Basic to the Macintosh world. These enhancements include programmable windows, menus, scrolling, sound and sound effects, compatibility with MacWrite, complete mouse control,

editing, debugging, access to most if not all of the Mac internal ROM routines, and elimination of the need for line numbers. This is what I call programming made easy. From what I've seen so far, there will be quite a competition between MacBasic and Microsoft Basic 2.0 in the coming months.

The Mac Basic listing shows an example of the ease in documenting a program as it is written. I originally wrote the program in Microsoft Basic version 1.0 as an educational tool for my 2 year old to learn his alphabet. The program prints the alphabet and then prompts the user to find each letter one at a time. Then the user points to the letter with the mouse and presses the mouse button. By using variable names that show what is happening and with the use of line labels, the program requires little documentation. When I write programs, I rarely have the time to finish up with documentation, especially when my 2 year-old can't wait to start the game. The use of these labels will be familiar to anyone who has used Basic on the Hewlett Packard 9826-36 series computers. The label name is used in each of the gosub statements in place of the line numbers. This makes the use of line numbers unnecessary. The corresponding label name also appears at the beginning of the subroutine followed by a colon. By indenting the text of the program within each subroutine, it is clear where the subroutine starts and ends and the label gives a clue as to what each subroutine does.

Next month we will explore Basic programming on the Mac in more detail.

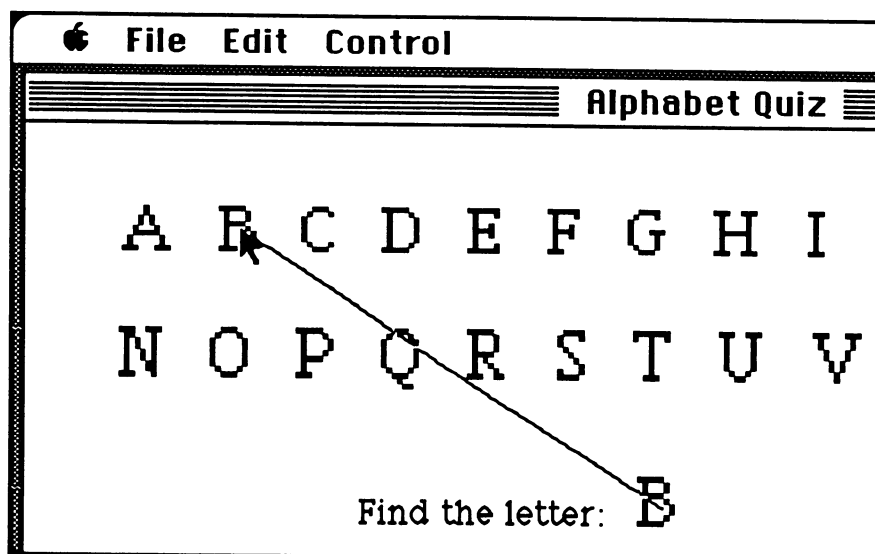


Fig. 1 Output from Basic program

```

10 *****
20 ' Alphabet Soup
30 ' By David Kelly
40 ' MS Basic Version 1.0
70 *****
1000 DIM TOP%(26),BOT%(26),RGT%(26),LFT%(26)
1010 FOR LETTER%=1 TO 26
1020 READ LFT%(LETTER%), TOP%(LETTER%),
 RGT%(LETTER%), BOT%(LETTER%)
1030 NEXT LETTER%
1040 GOSUB 1090: 'Print_alphabet
1050 FOR LETTER%=ASC("A") TO ASC("Z")
1060 GOSUB 1200: 'Ask_for_letter
1070 NEXT LETTER%
1080 CALL TEXTMODE(0):CALL TEXTFACE(0):CALL
 TEXTFONT(1):CALL TEXTSIZE(12):END

1090 ' Print_alphabet to screen
1100 CLS : 'Clearwindow
1110 CALL TEXTFONT(2): ' Set to New York Font
1120 CALL TEXTSIZE(24): ' Set Font size to 24
1130 CALL TEXTMODE(2): 'Sets screen to XOR
1140 CALL TEXTFACE(0): 'Sets face to plain style
1150 FOR LETTER% = 1 TO 26
1160 CALL
MOVETO(LFT%(LETTER%),BOT%(LETTER%)):PRINT
 CHR$(ASC("A")-1+LETTER%)
1170 NEXT LETTER%
1180 CALL TEXTSIZE(12): 'Set font size to 12
1190 RETURN

1200 ' Ask_for_letter
1210 CALL MOVETO (125,175):PRINT "Find the letter:
";
1220 CALL TEXTSIZE(24): 'Set Font size to 24
1230 CALL TEXTMODE(0)
1240 PRINT " ";CHR$(LETTER%)
1250 CALL TEXTSIZE(12)
1260 CALL PENMODE(10)

1270 IF MOUSE(0)=0 THEN 1270: 'button wait
1280 X2=MOUSE(1): Y2=MOUSE(2)
1290 CALL MOVETO(238,168):CALL LINETO(X2,Y2)
1300 IF MOUSE(0)<0 THEN 1300
1310 IF (X2<LFT%(LETTER%-64) OR
X2>RGT%(LETTER%-64) OR Y2< TOP%(LETTER%-64)
OR Y2 >BOT%(LETTER%-64)) THEN FLAG%=1 ELSE
FLAG%=0

1320 CALL PENMODE(14):PATTERN%(0)=0:CALL
PENPAT(VARPTR(PATTERN%(0)))
1330 CALL MOVETO(238,168): CALL
LINETO(X2,Y2):CALL PENNORMAL
1340 CALL TEXTMODE(0):IF FLAG% =1 THEN 1210
1350 BEEP:BEEP
1360 RETURN

```

(The following data statements are required by both versions, but are given here for MS Basic.)

```

1370 DATA 36,20,52,74 :A data
1380 DATA 72,20,84,74 :B data
1390 DATA 103,20,115,74 :C data
1400 DATA 133,20,147,74 :D data
1410 DATA 165,20,176,74 :E data
1420 DATA 195,20,207,74 :F data
1430 DATA 225,20,239,74 :G data
1440 DATA 257,20,272,74 :H data
1450 DATA 290,20,315,74 :I data
1460 DATA 330,20,350,74 :J data
1470 DATA 369,20,383,74 :K data
1480 DATA 401,20,414,74 :L data
1490 DATA 434,20,454,74 :M data
1500 DATA 35,75,51,120 :N data
1510 DATA 69,75,83,120 :O data
1520 DATA 101,75,115,120 :P data
1530 DATA 133,75,147,120 :Q data
1540 DATA 165,75,179,120 :R data
1550 DATA 197,75,208,120 :S data
1560 DATA 227,75,240,120 :T data
1570 DATA 259,75,275,120 :U data
1580 DATA 294,75,310,120 :V data
1590 DATA 329,75,352,120 :W data
1600 DATA 371,75,385,120 :X data
1610 DATA 402,75,417,120 :Y data
1620 DATA 434,75,445,120 :Z data

```

```

| *****
| Alphabet Soup *
| by Dave Kelly *
| MacBasic 0.82 version *
| *****

DIM Top%(26), Bot%(26), Rgt%(26), Lft%(26)
SET OUTPUT
FOR Letter% = 1 to 26
 READ Lft%(Letter%), Top%(Letter%),
 Rgt%(Letter%), Bot%(Letter%)
NEXT Letter%
GOSUB Print_alphabet
FOR Letter% = ASC ("A") TO ASC ("Z")
 GOSUB Ask_for_letter
NEXT Letter%
GTEXTNORMAL : END

Print_alphabet: CLEARWINDOW
 SET FONT 2 ! Set to New York Font
 SET FONTSIZE 24 ! Set Font size to 24
 SET GTEXTMODE 10 ! Sets screen to XOR
 SET GTEXTFACE 0 ! Sets face to plain
style
 FOR Letter% = 1 to 26
 GPRINT AT Lft%(Letter%), Bot%(Letter%);
 CHR$ (ASC ("A")-1+Letter%)

```

```
NEXT Letter%
SET FONTSIZE 1 ! Set Font size to 12
RETURN
```

Ask\_for\_letter: SET FONT 2

```
GPRINT AT 125,175;"Find the letter: ";
SET FONTSIZE 24 ! Set Font size to 24
SET GTEXTMODE 8 ! Set screen to cover mode
GPRINT " "; CHR$(Letter%)
SET FONTSIZE 12
SET PENMODE 10
BTNWAIT
X2 = MOUSEH : Y2 = MOUSEV
PLOT 238,168; X2,Y2
```

```
IF (X2<Lft%(Letter%-64) OR X2>
Rgt%(Letter%-64) OR Y2< Top%(Letter%-64) OR
Y2 > Bot%(Letter%-64)) THEN
 Flag%=1
ELSE
 Flag%=0
ENDIF
PLOT 238,168; X2,Y2
PENNORMAL : SET GTEXTMODE 8
IF Flag%=1 THEN GOTO Ask_for_letter
SOUND : SOUND
RETURN
```



# BASIC SCHOOL

## Does Anybody Really Know What Date It Is?



Article by Dave Kelly  
Program by Michael Steiner  
MacTutor Vol. 1 No. 2

REAL standalone (almost) applications can now be written in Basic. Microsoft Basic version 2.0 includes the required routines to make your Macintosh look just like those fancy assembly language application programs. This month's feature includes a useful example of what Microsoft Basic 2.0 can do. After waiting months for the Mac development tools we need, we can now use them to keep track of all those months with a perpetual calendar.

The program written by Michael Steiner will give you a quick overview of some of the features of Microsoft Basic 2.0. I found that I preferred editing the Basic listing with MacWrite and saving the file as text. Microsoft Basic will read MacWrite text files and MacWrite will read Microsoft Basic programs stored as text.

Anyone who has used the previous versions of Microsoft Basic will be glad to know that programs they have written for version 1.0 will work on version 2.0, however they won't take advantage of the extended features without modification. The most noticeable feature is the ability to control all the pulldown menus.

Type in the program and run it. The program opens a window to full screen size and then erases all the menus used by Basic. Then a second window opens up and displays instructions. The "OK" button was programmed with the one line **BUTTON** command. **DIALOG(0)** and **DIALOG(1)** return information about the status of the button or buttons. With a few simple lines you can design your own dialog windows and set up buttons to select various options.

**OK**

After clicking the "OK" button, the program displays a set of options available using the various button styles available. If you don't want to print a whole year you can deselect the "Entire year" option. The programmed menus let you select the beginning and ending months to print. You can program your own menus by using the **MENU** command. By issuing the appropriate parameters with the menu command you can enable or disable various menu options. **MENU(0)** and **MENU(1)** provide the means to poll the status of the menu selection. We will explore the **MENU** command in depth in future columns.

Notice that you can change the date by simply clicking and editing the date in the year box. All the Macintosh editing functions are easily invoked. In fact, the year box works just like the type and creator boxes in the Set File utility. Obviously Basic 2.0 can create applications that appear just like standalone programs. This language may be even more powerful than Apple's Mac Pascal.

After selecting the options you want, click the "OK"

button and the program prints each month to the screen one at a time. If you selected the "Show Julian" option, the Julian date will be printed along with the calendar. There is no option to print the calendar on the printer, but you can use the command-shift 4 option to print the window or modify the program yourself to print to the printer. Another feature you can try adding is to create another button for scrolling the calendar back and forth. Be sure to share any ideas you come up with for Basic with our readers by writing me care of MacTech. Happy New Year!

\*\*\*\*\*

### Perpetual Calendar

'Copyright 1984, Michael Steiner, All Commercial Rights Reserved 26 Oct 84

'Permission is granted to use, copy, and modify this program for your own personal use

'Modified and comments by David Kelly

\*\*\*\*\*

**WINDOW 1,"", (0,12)-(512,342),3** 'Open full screen window  
**sm=1:em=12** 'Initial start and end months  
**oldsm=sm:oldem=em**

**DIM status (2,12)**

**FOR i = 5 TO 1 STEP -1** 'Erase all the basic menus

**MENU i,0,0,""**

**NEXT i**

instructions:

**WINDOW 2,"", (3,70)-(509,270),3** 'Open 2nd window  
**WIDTH 62**

**PRINT**

**PRINT**"This program will generate a calendar for any year from 1753 onward."

**PRINT**"You may choose whether you want to display Julian Dates as well as the days of the month. You may select to print the whole year or you may choose starting and ending months."

**PRINT:PRINT**"Click OK when you have finished reading these instructions."

**BUTTON 1,1,"OK", (210,150)-(250,170),1** ' Set up button #1  
**WHILE DIALOG(0) <> 1 :WEND** ' Wait for button #1

setup:

**WINDOW 2,"", (50,100)-(450,200),2** 'Change 2nd window and open it

**TEXTFACE (1)**

**MOVETO 250,15:PRINT**"Show Julian":**MOVETO 250,30:PRINT** "Date?"

**BUTTON 1,1,"Yes", (250,40)-(300,60),3** 'Set up buttons

**BUTTON 2,2,"No", (250,65)-(290,85),3**

**BUTTON 3,2,"Print Entire year", (50,60)-(200,80),2**

**BUTTON 4,1,"OK", (360,60)-(390,80),1**

```

MOVETO 50,40:PRINT"Year"
TEXTFACE (0)
EDIT FIELD 1,RIGHT$(DATE$,4), (90,30)-(130,45),2
'Set up Edit field with current year
DATA January, February, March, April, May, June, July,
August, September, October, November, December
DIM month$(12)
FOR i = 1 TO 12:READ month$(i):NEXT i 'Read the months
BStatus=1:GOSUB EntireYear
DialogActive = 1
WHILE DialogActive
 EventType = DIALOG(0) 'See if a button is pressed
 IF EventType = 1 THEN GOSUB ButtonEvent
 MenuId = MENU(0) 'See which menu is selected
 IF MenuId THEN GOSUB ChooseMonth
WEND
Year = VAL(EDIT$(1)):IF Year < 1753 THEN Year = 1753
'Get year data
WINDOW CLOSE 2

```

```

calendar:
MENU 1,0,0,"" 'Clear menu 1
MENU 2,0,0,"" 'Clear menu 2
BUTTON 1,1,"Continue", (50,305)-(125,325),1 'Set up
button 1
BUTTON 2,1,"Quit", (350,305)-(400,325),1 'Set up button 2
TEXTSIZE(12):TEXTFACE(0):TEXTMODE(1) 'Set font
attributes
DATA 31,28,31,30,31,30,31,31,30, 31,30,31
DIM dm(12):FOR mo=1 TO 12: READ dm(mo):NEXT mo
'Read # of days/month
DIM M$(12):FOR MO=1 TO 12: M$(MO)=month$(MO)
:NEXT MO
DATA SUN,MON,TUE,WED,THU,FRI,SAT
DIM DAY$(7):FOR DAY = 1 TO 7: READ
DAY$(DAY):NEXT DAY 'Read days of week
DM(2)=DM(2)-(YEAR/4=INT(YEAR/4))+(YEAR/100=
INT(YEAR/100))-(YEAR/400=INT(YEAR/400)) 'Add a day
for leap year
DIM jd(12)
FOR i=1 TO 12:jd(i)=jd(i-1)+dm(i-1):NEXT i
Y=INT(((365.25*YEAR+jd(sm))/7)-1.2499999#)
IF Y<0 THEN Y=Y+7
CALL PENSIZE (3,3)
FOR M=sm TO em
 dy=jd(m)
 CLS' Draw the calendar
 FOR I = 1 TO 7:LINE (5,I*40+10)-(481,I*40+10):NEXT I
 FOR I = 0 TO 7:LINE (I*68+5,50)-(I*68+5,290):NEXT I
 TEXTSIZE (24):TEXTFACE (17)' Set calendar font
attributes
CALL MOVETO (130,34):PRINT M$(M);" ";YEAR;
TEXTSIZE (12):TEXTFACE(1)
FOR I= 0 TO 6:CALL MOVETO (I*68+10,49):PRINT
DAY$(I+1):NEXT I
FOR DM = 1 TO DM(M)
 TEXTFACE(9):TEXTSIZE(18)
 CALL MOVETO (Y*68,X+71):PRINT DM:IF JU THEN
GOSUB JulianPrint
 Y=Y+1:IF Y>6 THEN Y=0:X = X+40
NEXT dm

```

```

WHILE DIALOG(0) <> 1 :WEND 'Wait for button press
IF DIALOG(1)=2 THEN M=em
X=0
NEXT M

```

```

EndRoutine: 'Set up End menu
WINDOW CLOSE 1
MENU 1,0,1,"Options"
MENU 1,1,1,"Re-run Program"
MENU 1,2,1,"Exit to Finder"
WHILE MENU(0) = 0:WEND 'Wait till selection is made
IF MENU(1)=1 THEN RUN : ELSE SYSTEM 'Return to
Finder

```

```

ButtonEvent:
ButtonId = DIALOG(1)
IF ButtonId=4 THEN DialogActive=0
ON ButtonId GOSUB Julian, NoJulian, EntireYear
RETURN

```

```

Julian: 'Set button for Julian year print
BUTTON 1,2
BUTTON 2,1
JU=-1
RETURN

```

```

NoJulian: 'Set button for Julian no print
BUTTON 1,1
BUTTON 2,2
JU=0
RETURN

```

```

EntireYear: 'Set beginning month to Jan. & Ending month to
Dec.
IF BStatus = 1 THEN BStatus =2:sm=1:em=12:ELSE
BStatus = 1
FOR ms = 1 TO 2:FOR st= 0 TO 12:
status(ms,st)=ABS((BStatus=1)):NEXT st,ms
BUTTON 3,BStatus
MENU 1,0,status(1,0),"Starting Month"
FOR i= 1 TO 12
 MENU 1,i,status (1,i),month$(i)
NEXT i
MENU 1,sm,2,month$(sm)

```

```

Endmonth:
MENU 2,0,status(2,0),"Ending Month"
FOR i = 1 TO 12
 MENU 2,i,ABS(sm<(i+1)),month$(i)
NEXT i
IF em >= sm THEN MENU 2,em,2,month$(em)
RETURN

```

```

ChooseMonth:
ItemId = MENU(1)
IF MenuId=1 THEN oldsm=sm: sm=ItemId:MENU 1,0,1:
MENU 1,sm,2,month$(sm):IF oldsm<>sm THEN MENU
1,oldsm,1,month$(oldsm)
IF MenuId=1 THEN GOSUB EndMonth
IF MenuId=2 THEN oldem=em: em=ItemId:MENU 2,0,1:
MENU 2,em,2,month$(em):IF oldem<>em OR oldem> sm
THEN MENU 2,oldem, ABS(oldem>=sm),month$(oldem)

```

```
IF em<sm THEN em=sm: MENU 2,em, 2,month$(em)
RETURN
```

JulianPrint: 'Print Julian year

```
dy=dy+1
```

```
dy$=RIGHT$(STR$(year),1)+
```

```
RIGHT$("00"+MID$(STR$(dy),2),3)
```

```
dy=VAL(dy$)
```

```
TEXTSIZE (12):TEXTFACE (0)
```

```
CALL MOVETO (Y*68+10,X+85)
```

```
PRINT dy$;
```

```
RETURN
```



# Basic School

## May I Have Your Order, Please?



Dave Kelly  
MacTech Editorial Board  
MacTutor Vol. 1 No. 3

This month's BASIC School features programming menus on the MAC. One of the first things I do when I start a program is the menu. This way I'm sure that I know what I want to do in the program and it helps me to organize in a "top-down" fashion. Hopefully, after reading this you will know the basics of menu programming. The best thing is to experiment and test your own programs out using ideas you may gain from reading here.

First a bit of a rundown on the syntax for Programming menus in Microsoft BASIC (version 2.0). The statements we have at our disposal are:

- MENU menu-id, item-id, state [,title-string]
- MENU
- MENU RESET

*Menu-id* is a number from 1 to 10 representing the menu bar selected. *Item-id* is a number from 1 to 20 representing the menu item selected. If *Item-id* is 0, the entire menu is selected.

The words *menu-id* and *item-id* make good descriptive variable names for the functions they perform. Use 0 for *state* to disable the menu or menu item, 1 to enable it or 2 to enable it and place a check mark by it. The *title-string* (optional) is a string representing the title of the selected menu bar or the selected item in the menu.

The MENU command used by itself returns the current menu selection to normal black-on-white video. MENU RESET is used to set all the menu bars back to the menus used by BASIC. It is also a way to erase your custom menus when returning back to basic from a program.

Two more functions (called function syntax) are used to poll the status of the menu selection from within a program:

- MENU(0)
- MENU(1)

MENU(0) returns a number corresponding to the number of the last menu bar selection. Once it is executed,

MENU(0) is reset to 0. MENU(1) returns a number which corresponds to the number of the last menu item selected. The function syntaxes are executed by setting the function equal to a variable (example: Menuid = MENU(0)). In most of the sample programs I have seen, the variables Menuid and Itemid are used for MENU(0) and MENU(1) respectively. Any legal variable name could be used, but these seem to describe the functions quite well.

OK, now we know all about menu commands. The example program was written to demonstrate the menu functions. I was hungry at the time I started writing the program, so you can see how it came out. The program sets up a lunch "menu" and asks you to select what you would like. If there isn't anything you like, the program could be modified to include just the things you like or even your own routines. Then when you have made whichever combination of lunch selections you may then "Eat it". After that you may return to make more selections, but only those items which have not been previously selected. When everything has been eaten or you are tired of eating, you may stop the program and return to BASIC.

Now a more detailed explanation of how the menu options are controlled in the program. The first 9 menu statements set up my custom menu. Since this is the first time that each item has been used I have given each menu item a name. If no name had been specified, the menu item would have been blank. To erase previous menu items you can use "" (Null) as the *title-string* option in each menu item you want to erase. Don't erase menus you may need later. It is better to deselect the menu or menu item with the *state* argument.

The *menu-id* is set to 6 because I use the 6th menu bar in order to preserve the BASIC menus. The BASIC menus could be written over and the new menu would take its place. The second argument, *item-id*, is numbered from 0 to 8, representing a different menu item. (A zero represents the entire menu). The *state* is set to 1 to enable each menu item. After some setup of variables, the statement ON MENU

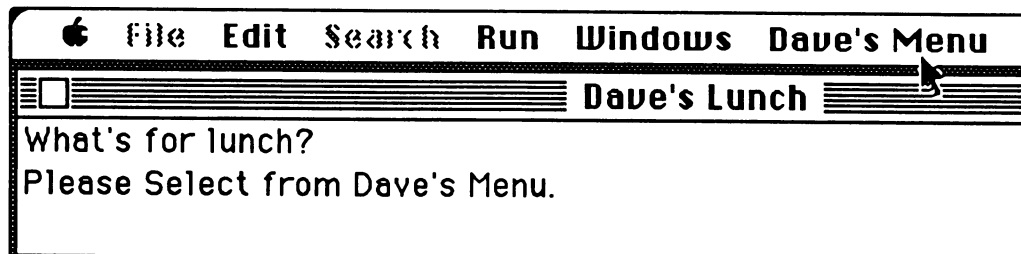


fig. 1

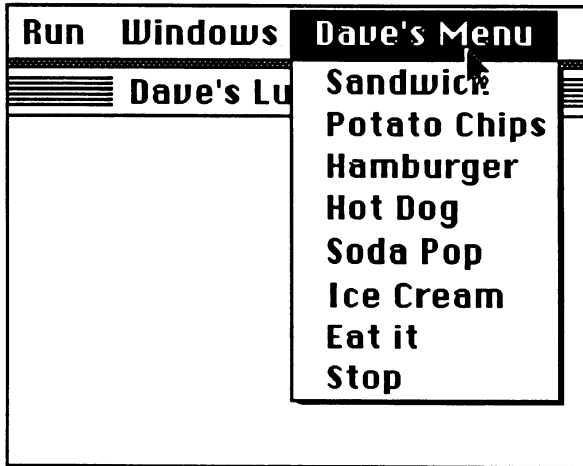


fig. 2

GOSUB enables event trapping for the menus. Event trapping will be covered at a later time, however for this case, this statement indicates what will happen as soon as any menu is selected. In this case, the program will jump to the subroutine named 'loop'. However, this event trapping is not enabled until the statement, MENU ON is executed. After the MENU ON statement the program goes into an infinite loop where it will wait for a menu to be selected. You should especially be careful when in infinite loops to be sure that there will eventually be some way to get out of the loop. If you had erased the BASIC menus and then somehow become stuck in a loop someplace, the only way out of it is the command-". Beware when error trapping the break key (command-".) using the BREAK statements, there may be no way out of an endless loop without doing a system reset or turning off the power.

The program then jumps to the 'loop' routine. There the variables, MenuId and ItemId are set equal to the functions MENU(0) and MENU(1) so that the program can tell which of the menu items was selected. The ON ItemId GOSUB statement specifies which subroutine the selection will cause the program to branch to.

When the program begins, all of the items are selected, but none have a check mark by them. Take a look at one of the subroutines, say the starting at the label titled "Hamburger". Using labels makes it clear which routine does what, especially if the labels describe what the routine is about. All the routines are similar and will show the same technique. The variable 'Hamburger.select' is set to 1 to indicate that the menu is active, with no check mark. If the menu item is selected, the 'Hamburger.select' variable is toggled between 1 and 2 each time it is selected (unless the menu item has been disabled by setting the variable to 0. The MENU 6,3,Hamburger.select statement sets up the condition of the menu, 0 for disabled, 1 for enabled (no checkmark), 2 for enabled with checkmark. The 'Eat it' subroutine will disable any menus which have been enabled with the checkmark. This makes it impossible to eat more than one hamburger. (Fortunately, there are still a lot of other things to eat.) If no items have been checked and 'Eat it' is selected from the menu,



fig. 3

the entire menu is returned to the normal black-on white video with the MENU statement before exiting from the 'Eat it' routine.

After making the menu selection, the program branches back to the infinite loop where it started before being "event trapped" by the menu selection. When 'Stop' is selected, the program initiates the MENU RESET statement and returns to BASIC.

Since this program was designed for study (it serves no real function, especially since it isn't edible), I would advise that you use the trace and step functions to see just how each statement affects the program. If you want to get really creative you could have the program draw the food for you after each selection.

The program uses the method of looping in an infinite loop while waiting for a menu selection to interrupt the loop. Another method which takes much more caution when writing the program involves using Event Trap Programming. This is explained briefly starting on page 68 of the Microsoft BASIC 2.0 manual (Advanced topics). Three commands, MENU ON, MENU OFF, MENU STOP are used liberally to determine when menus can be accessed and what events can occur when they are accessed. The problems occur when several menus call the same routines and use the same variables. When the menu is selected and event trapping is enabled, the menu will interrupt the program wherever it is and go run the selected routine, possibly changing variables that were being used when the program was interrupted. Event Trap Programming will be covered at a later time, hopefully when more of the basics of programming Macintosh BASIC have been covered. Next time we will open up some windows and explore some more keys to Programming BASIC on the MAC.

-----  
Dave's Lunch Menu  
by Dave Kelly  
MACTECH © February 1985  
-----

MENU 6,0,1,"Dave's Menu" 'New menu  
MENU 6,1,1,"Sandwich"  
MENU 6,2,1,"Potato Chips"



```

MENU 6,3,1,"Hamburger"
MENU 6,4,1,"Hot Dog"
MENU 6,5,1,"Soda Pop"
MENU 6,6,1,"Ice Cream"
MENU 6,7,1,"Eat it"
MENU 6,8,1,"Stop"
Sandwich.select=1:Potato.select=1:
 Hamburger.select=1:Hot.select=1
 Soda.select=1:Ice.select=1:count=0
PRINT "What's for lunch?"
PRINT"Please Select from Dave's Menu."
ON MENU GOSUB loop
MENU ON
Pause:GOTO Pause 'Wait for menu selection in infinite loop
Loop:
MenuId = MENU(0) 'Menu # selected
ItemId = MENU(1) 'Item # selected
ON ItemId GOSUB Sandwich,
Potato.Chips,Hamburger,Hot.Dog,
Soda.Pop,Ice.Cream,Eat.it,Stopit
RETURN
Sandwich:
IF Sandwich.select=1 THEN Sandwich.select=2:GOTO
Set.Sandwich
IF Sandwich.select=2 THEN Sandwich.select=1
Set.Sandwich:MENU 6,1,Sandwich.select
RETURN
Potato.Chips:
IF Potato.select=1 THEN Potato.select=2:GOTO Set.Potato
IF Potato.select=2 THEN Potato.select=1
Set.Potato:MENU 6,2,Potato.select
RETURN
Hamburger:
IF Hamburger.select=1 THEN Hamburger.select=2:GOTO
Set.Ham
IF Hamburger.select=2 THEN Hamburger.select=1
Set.Ham:MENU 6,3,Hamburger.select
RETURN

```

```

Hot.Dog:
IF Hot.select=1 THEN Hot.select=2:GOTO Set.Hot
IF Hot.select=2 THEN Hot.select=1
Set.Hot:MENU 6,4,Hot.select
RETURN
Soda.Pop:
IF Soda.select=1 THEN Soda.select=2:GOTO Set.Soda
IF Soda.select=2 THEN Soda.select=1
Set.Soda:MENU 6,5,Soda.select
RETURN
Ice.Cream:
IF Ice.select=1 THEN Ice.select=2:GOTO Set.Ice
IF Ice.select=2 THEN Ice.select=1
Set.Ice:MENU 6,6,Ice.select
RETURN
Eat.it:
PRINT:PRINT"You have just eaten:"
old.count=count
IF Sandwich.select=2 THEN Sandwich.select=0: MENU
6,1,0:PRINT "Sandwich":count=count+1
IF Potato.select=2 THEN Potato.select=0: MENU
6,2,0:PRINT"Potato Chips":count=count+1
IF Hamburger.select=2 THEN Hamburger.select=0:MENU
6,3,0: PRINT"Hamburger":count=count+1
IF Hot.select=2 THEN Hot.select=0:MENU 6,4,0:
PRINT"Hot Dog":count=count+1
IF Soda.select=2 THEN Soda.select=0:MENU
6,5,0:PRINT"Soda Pop":count=count+1
IF Ice.select=2 THEN Ice.select=0:MENU 6,6,0:PRINT"Ice
Cream":count=count+1
IF count=6 THEN MENU 6,7,0:PRINT"The food is all gone."
IF old.count=count THEN PRINT"Nothing":MENU
Deselect current menu selection
RETURN
Stopit:MENU RESET 'reset the BASIC menu bars and
then END the program
END

```

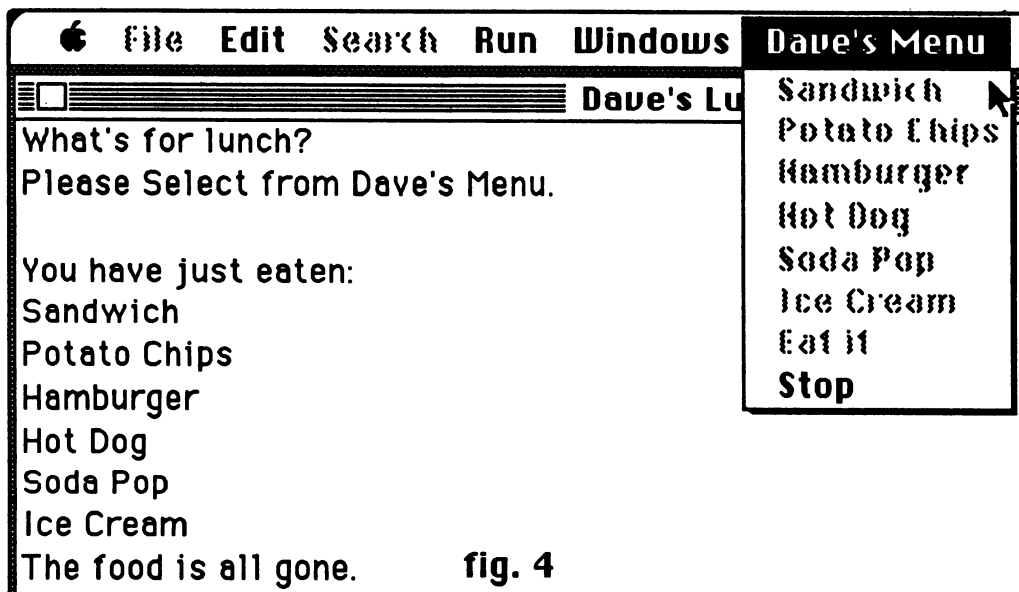


fig. 4

# Basic School

## What Light Through Yonder Window Breaks



Dave Kelly  
MacTech Editorial Board  
MacTutor Vol. 1 No. 4

Let's put some light on the subject of windows. Almost everything you write in Microsoft BASIC 2.0 requires some window programming. This is especially true if you want your program to conform to the Macintosh user interface. First, a rundown on the commands we have available to program our windows:

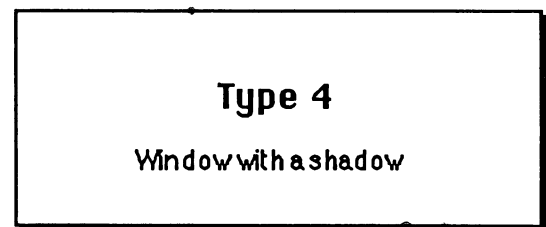
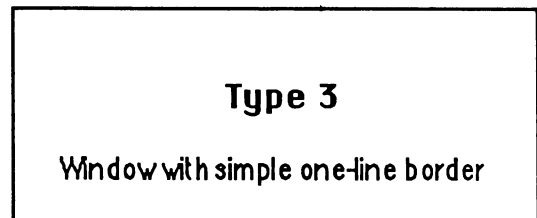
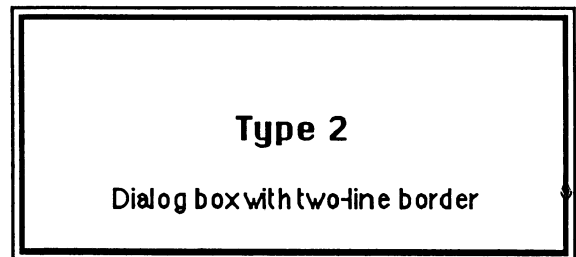
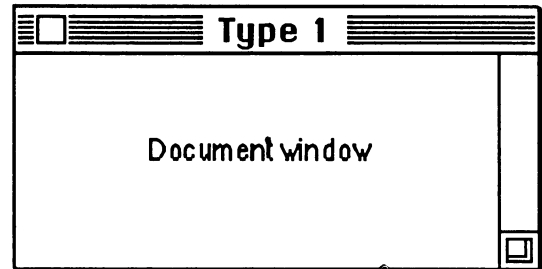
```
WINDOW window-id [title]
 [, [rectangle] [, type]]
WINDOW CLOSE window-id
WINDOW OUTPUT window-id
WINDOW OUTPUT #file-number
```

The *window-id* identifies the output window. *Window-id* is a number from 1 to 4. This means that you may only have up to 4 windows open at any one time. BASIC opens up Window 1 when BASIC starts. If this window gets closed you can select the output window from the Windows menu in BASIC or type **WINDOW 1** from the command window. A window **must** be open and specified as the output window before anything can be printed in the window.

The *title* is the name displayed in the title bar of a document window if the window has a title bar.

The *rectangle* specifies the location of the window on the Macintosh screen. It is given in the form (x1,y1)-(x2,y2) where (x1,y1) is the upper left corner of the window and (x2,y2) is the lower right corner of the window. This sounds easy but someone out there will probably be confused. The entire Macintosh screen has a resolution of 512 X 342. To open a window which fills the entire screen you could use (0,0)-(512,342). However, you'll find that the desktop border (for pull down menus will cover the title bar of your document window. The desktop border on the top of the screen is about 40 pixels down from the top. By using (2,40) as the top corner and (510,340) as the bottom corner your window will fill the screen except for the desktop border. A couple of pixels are allowed at the edge of the screen to make it look a little better. So much for filling the entire screen with a window. It can require some trial and error to get your window positioned where you want when you try to center a smaller window on the screen. Other BASIC commands use the *rectangle* format to specify the location relative to the upper-left corner of the window. The *rectangle* format in the **WINDOW** command specifies the location of the window relative to the upper-left corner of the screen.

There are 4 types of windows that are available with *type*. They are (for *type* = 1-4) :



Only *type* 1, the document window, has a size box and a title bar. Also, only *type* 1 can be moved with the mouse. The other types will not change position on the screen without redefining the window with the **WINDOW** command. A negative type number is called a modal dialog box. In this case any attempt to select outside of the window results in a beep.

The **WINDOW CLOSE** statement closes the indicated window. **WINDOW OUTPUT** indicates which window will be the window used for output when there are multiple windows. This allows you to send output to another window without changing the active window. If a *file-number* is indicated instead of a *window-id*, the output will be sent to the indicated file. Currently, only files opened to LPT1: are

valid with this statement. As other devices which support graphics are made available, this will change.

## INFO ABOUT WINDOWS

The window function **WINDOW(*n*)** returns information about the windows. If *n* = 0, the function returns the *window-id* of the active output window. *n* = 1 returns the *window-id* of the current output window. This is the window where output will be sent. *n* = 2 gives the width of the current output window and *n* = 3 returns the height of the current output window. *n* = 4 and *n* = 5 returns the x and y coordinate (respectively) in the current output window where the next character will be drawn. This function may be used to determine where the next character or section of a drawing should continue.

Our sample window shows how these commands are used to control windows. The program sets up all four windows as type 1 (document with size box and title). Information about each window is printed in the window. The information about the width and height might be used when trying to get just the right sized window for your program.

An important thing to note when working with windows is that everything does not happen automatically as you may be used to when moving, opening, or closing windows in application programs you may have run. You have to do everything which you took for granted before. For some people this may be hard to figure out, but just try to imitate the way the commercial applications are supposed to use the Macintosh user interface. Try this with my sample program: move one of the windows on top of another and then back again. The printed information which may have been destroyed by overlaying a window over another needs to be refreshed. Changing the size box has the same effect.

## REFRESHING WINDOWS

The **DIALOG(5)** function returns the *window-id* of the window which needs to be refreshed. Updating a window is not automatically performed but must be done by the program. In the *Event:* subroutine when **DIALOG(0)** sets *d*=5, the output window is set to the window indicated by **DIALOG(5)** and the window is printed again. It would be desirable to setup a subroutine to refresh windows if this feature is desired. Just for fun, delete the line that starts with **IF d=5 THEN WINDOW OUTPUT DIALOG(5)** and then run the program. You can see that the window will not be automatically refreshed without this event trapping. In some programs you may not want to refresh all (or even any) of the windows.

The *printwindowinfo:* subroutine prints the information on the current output window. You can change this and print your own message if you like. Try changing the window type just to see what it will do.

The command **LOCATE[*row*][*column*]** positions the pen at a specified row and column in the output window. This position is with respect to the upper-left corner of the current

output window. Since most fonts are proportionally spaced, the spacing is that of the letter "O", because it is an average width for most fonts. Another way to control the location of the pen is the use of **CALL GETPEN**, **CALL MOVETO(*x*,*y*)** or **CALL MOVE(*xdelta*,*ydelta*)**. (Note: MSBASIC 2.0 allows you to optionally leave off the word **CALL** and just use the routine name: useful for creating your own basic commands by calling machine language routines).

Notice that by selecting the active window from the pull down menu causes a different result than clicking on the window. When the window is clicked it is selected as both the active and output window. By selecting from the menus, one window can be active while output is going to another. This is exactly the same procedure used when refreshing the screen. This is observed by changing the output window from the menu. To return to BASIC, use the quit menu.

You now have an introduction to windows. The important thing to remember is that you must control where windows are opened, and which windows are selected or deselected, and which window is the output window. For simple programs which only need a place to print, the statement **WINDOW** creates an output window if none currently exists and makes it active. Your program will have to do the work of deciding what should be done with each window, as it is not automatic. Do you have some questions or programming tips to share with MacTutor's readers? Send your questions or contributions to BASIC SCHOOL care of MacTutor, P.O. Box 400, Placentia, CA. 92670.

Window Demo

by Dave Kelly

MACTUTOR March 1985

```
'SET UP MENUS
MENU 4,0,1,"Quit"
MENU 4,1,1,"Return to BASIC"
MENU 5,0,1,"Output Window"
MENU 5,1,1,"Window 1"
MENU 5,2,0, "-" 'MENU DASH LINE
MENU 5,3,1,"Window 2"
MENU 5,4,0, "-"
MENU 5,5,1,"Window 3"
MENU 5,6,0, "-"
MENU 5,7,1,"Window 4"
MENU 6,0,1,"Active Window"
MENU 6,1,1,"Window 1"
MENU 6,2,0, "-"
MENU 6,3,1,"Window 2"
MENU 6,4,0, "-"
MENU 6,5,1,"Window 3"
MENU 6,6,0, "-"
MENU 6,7,1,"Window 4"
```

```
'TURN ON MENU SELECT TRAP
ON MENU GOSUB menus:MENU ON
```

```
'SET UP WINDOWS and PRINT INFO
```

```
WINDOW 1,"Window 1",(2,40)-(252,190)
```

```

GOSUB printwindowinfo
WINDOW 2,"Window 2", (260,40)-(510,190)
GOSUB printwindowinfo
WINDOW 3,"Window 3", (2,195)-(252,345)
GOSUB printwindowinfo
WINDOW 4,"Window 4", (260,195)-(510,345)
GOSUB printwindowinfo
DIALOG ON

```

'ENABLE EVENT TRAP

ON DIALOG GOSUB Event  
 pause:GOTO pause

Event:

```

d=DIALOG(0) 'EVENT OCCURED
 'd=3 USER CLICKED INACTIVE WINDOW
IF d=3 THEN window.picked=
 DIALOG(3):WINDOW window.picked
 'd=4 USER CLICKED GO-AWAY BOX
IF d=4 THEN window.closed=
 DIALOG(4):WINDOW CLOSE window.closed
 'd=5 WINDOW NEEDS TO BE REFRESHED
IF d=5 THEN WINDOW OUTPUT
 DIALOG(5):GOSUB printwindowinfo
RETURN

```

menus: 'THIS HANDLES MENUS

```

menunumber=MENU(0)
menuitem=MENU(1):MENU
IF menunumber<>4 THEN menu5
 'MENU 4 QUILTS

```

#### MENU RESET:END

```

menu5:IF menunumber<>5 THEN menu6
 'MENU 5 SELECTS OUTPUT WINDOW

```

```

IF menuitem=1 THEN WINDOW OUTPUT 1
IF menuitem=3 THEN WINDOW OUTPUT 2
IF menuitem=5 THEN WINDOW OUTPUT 3
IF menuitem=7 THEN WINDOW OUTPUT 4
GOSUB printwindowinfo
RETURN

```

menu6:

```

IF menunumber<>6 THEN RETURN 'MENU 6 SELECTS
ACTIVE WINDOW
menuitem=MENU(1):MENU
IF menuitem=1 THEN WINDOW 1
IF menuitem=3 THEN WINDOW 2
IF menuitem=5 THEN WINDOW 3
IF menuitem=7 THEN WINDOW 4
GOSUB printwindowinfo
RETURN

```

printwindowinfo:

```

LOCATE 1,1
PRINT"Current active window is ";
 WINDOW(0)
PRINT"Current output window is ";
 WINDOW(1)
PRINT"Window width is ";WINDOW(2)
PRINT"Window height is ";WINDOW(3)
RETURN

```



# Basic School

## A Rectangle Utility for Controls & Edit Fields



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 5

Well, now that we know how to use windows, we should discuss what to put in them. Besides the usual printing of output, our Macintosh provides us with what are known as **BUTTONs** and **EDIT FIELDS**.

In MS Basic 2.0 the syntax is:

```
BUTTON button-id,state [,title,
 rectangle [,type]]
BUTTON CLOSE n
EDIT FIELD field-id [,default,
 rectangle [,type][,
 justify]]]
EDIT FIELD CLOSE field-id
```

The *id* parameter identifies the button or edit field number. **BUTTON** *states* are: 0=Button inactive, dimmed on the screen, 1=Button active, not selected, 2 =Button active, selected. *Title* is a string expression that is displayed inside or beside the identified **BUTTON**. The **BUTTON** *type* parameter identifies the type of **BUTTON**, 1=A simple push button, 2=A check box, 3=A radio button. The **EDIT FIELD** *default* is the string expression to be edited. Cut and paste editing can be used to edit the *default* string. *Type* indicates one of four editing formats. *Types* are: 1=Draw a box around the rectangle to be edited-Return keys not allowed, 2=Draw a box around the rectangle and allow return keys, 3=No box around edit field-Return keys not allowed, 4=No box around the rectangle and allow return keys. When using *types* 1 and 3 the edit field has wraparound (no return keys are allowed). The *justify* parameter specifies the justification of the text in the **EDIT FIELD**, 1=Left Justify, 2=Center Justify, 3=Right Justify. Now we're ready to program.

Well, almost. If you remember from our discussion of **WINDOWS**, the *rectangle* parameter specifies the location of the window on the Macintosh screen. The *rectangle* parameter in **BUTTON** and **EDIT FIELD** statements specifies the location of the **BUTTON** or **EDIT FIELD** in the current output window. It has the form (x1,y1)-(x2,y2), same as for **WINDOW**. (x1,y1) is the upper left corner of the current window and (x2,y2) is the lower right corner of the window. Sounds easy at first, but here's the catch: for every program you write you have to figure out where on the screen you want to put each **BUTTON** or **EDIT FIELD**. This can be quite a pain, especially if you have a lot to put in your window. Since there is no one *rectangle* for every application, you must be the judge of what you think "looks" good. This is true for all program development on the Mac. All of positions for the dialog boxes and window that we see in the finder and in other applications had to be decided the same way.

The program, *Rectangle Sizer* should be an aid to anyone experimenting with the positions and sizes of **BUTTONs** and **EDIT FIELDS**. When the program starts you may select **BUTTONs** or **EDIT FIELDS** from the Type Selection menu. The function menu allows you to add additional **BUTTONs** or **EDIT FIELDS** or erase. You may change the size of the output window and move it around, then select Window dimensions from the function menu and the program will print the current width and height of the output window. As you place **BUTTONs** or **EDIT FIELDS** in the window, the program will display the current *rectangle* coordinates. You may want to experiment and write down the coordinates for use in your program. If you want to use different fonts and sizes you can add a menu for that if you wish. This would be useful in programming the **EDIT FIELDS**. The **BUTTONs** always use 12-point Chicago. Therefore the *rectangle* size must be 12-point or more or part of the text will be clipped off. To use other fonts with **BUTTONs** you should print text beside the **BUTTON**. (You may have to move the desired fonts to your BASIC disk as only the bare minimum of fonts comes on the disk). It would take some experimenting to get things just right, however. You should experiment with the different **BUTTON** *types* and sizes to see what happens. A useful utility brought to you by **MACTUTOR**.

Rectangle Sizer  
By Dave Kelly  
MACTUTOR 1985

```
MENU 6,0,1,"Function"
MENU 6,1,0,"Add or Change Button #"
MENU 6,2,1,"Erase Button"
MENU 6,3,0,"-"
MENU 6,4,0,"Add or Change Edit Field #"
MENU 6,5,1,"Erase Edit Field"
MENU 6,6,0,"-"
MENU 6,7,1,"Window dimensions"
MENU 6,8,1,"Quit"
MENU 7,0,1,"Type Selection"
MENU 7,1,1,"Buttons"
MENU 7,2,1,"Edit Fields"
MENU 7,3,2,"No Selection"
rect%=0:bnumber%=0:enumber%=0
ON MENU GOSUB menuevent : MENU ON
WINDOW CLOSE 1
WINDOW 2,,(2,280)-(510,340),3: CLS
WINDOW 1,"Rectangle Sizer Window", (2,40)-(510,275)
ON MOUSE GOSUB mouseevent
'Watch for mouse click
MOUSE ON
pause:GOTO pause
```

| Windows    | Function                   | Type Selection |
|------------|----------------------------|----------------|
| angle Size | Add or Change Button #     |                |
|            | Erase Button               |                |
|            | Add or Change Edit Field # |                |
|            | Erase Edit Field           |                |
|            | Window dimensions          |                |
|            | Quit                       |                |

mouseevent:

```
MOUSE OFF: MENU OFF: x=MOUSE(0)
xstart=MOUSE(3): ystart=MOUSE(4)
```

loop:

```
x=MOUSE(0)
IF x>=0 THEN exitloop' Loop until button is released
xend=MOUSE(5):yend=MOUSE(6)
IF rect%=1 AND bnumber%>0 THEN BUTTON bnumber%,
State,Title$, (xstart,ystart)-(xend,yend),Type
IF rect%=2 AND (yend-ystart<=0) THEN yend=ystart+1
IF rect%=2 AND (xend-xstart<15) THEN xend=xstart+16
IF rect%=2 AND enumber%>0 THEN EDIT
FIELD enumber%, default$, (xstart,ystart)-(xend,
yend), EditType, Justify
CLS:WINDOW OUTPUT 2:LOCATE 1,1
IF rect%=1 THEN
PRINT"Rectangle";bnumber%; ELSE
PRINT "Rectangle";enumber%;
PRINT "is: (";xstart;",";ystart;")-(";xend;",";
yend;")":WINDOW 1
GOTO loop
```

exitloop:

WINDOW OUTPUT 2

```
LOCATE 1,1:IF rect%=1 THEN
PRINT"Rectangle";bnumber%; ELSE
PRINT "Rectangle";enumber%;
PRINT "is: (";xstart;",";ystart;")-(";xend;
",";yend;")": WINDOW OUTPUT 1
```

MOUSE ON: MENU ON: RETURN

menuevent:

menunumber=MENU(0)

IF menunumber=7 THEN menu7

IF menunumber<6 THEN RETURN

menuitem=MENU(1): MENU

ON menuitem GOSUB Changebutton, Erasebutton, blank,  
Changeedit, Eraseedit, blank, Windowsize, Quit  
RETURN

blank:RETURN 'This will never happen, but just in case...

menu7: menuitem=MENU(1)

IF menuitem=1 THEN MENU 7,3,1:MENU 7,1,2:

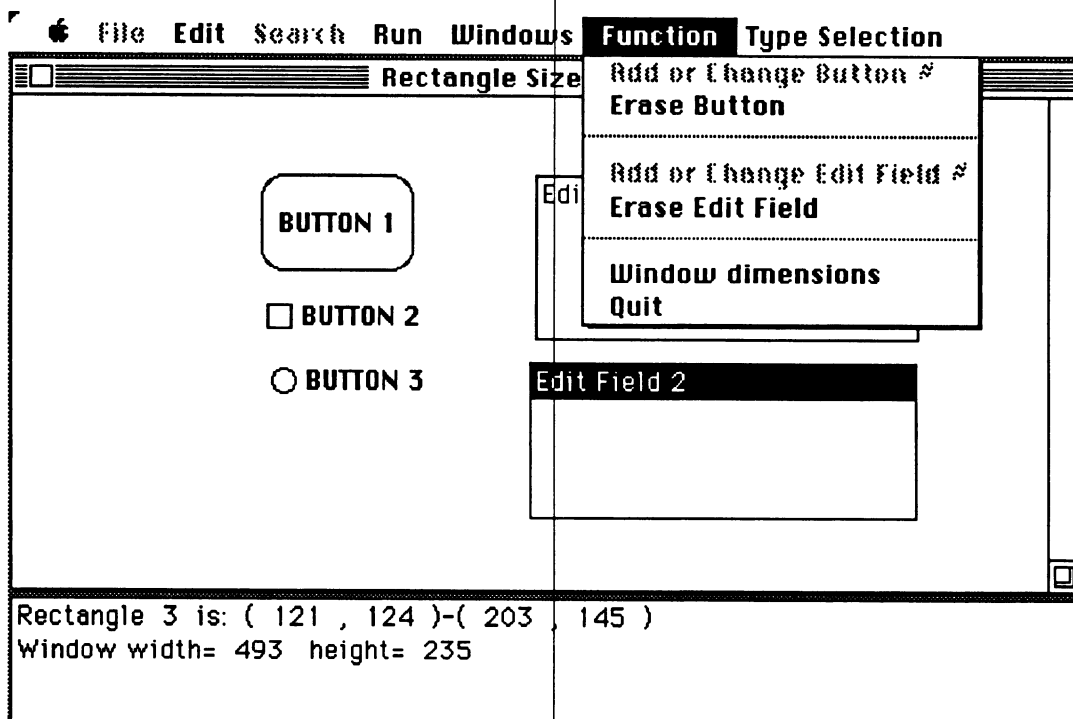
MENU 7,2,1: MENU 6,1,1: MENU 6,4,0: rect%=1:IF  
bnumber%=0 THEN GOSUB changebutton

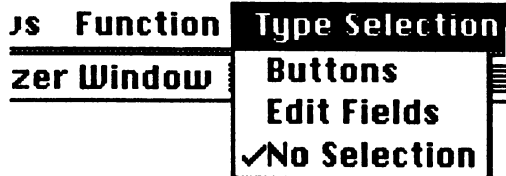
IF menuitem=2 THEN MENU 7,3,1:MENU 7,1,1: MENU  
7,2,2: MENU 6,1,0: MENU 6,4,1: rect%=2:IF enumber%=0  
THEN GOSUB changeedit

IF menuitem=3 THEN MENU 7,3,2:MENU 7,1,1: MENU  
7,2,1: MENU 6,1,0: MENU 6,4,0: rect%=0  
RETURN

Changebutton:

WINDOW 2: CLS:INPUT "Enter Button number: ",bnumber%  
IF bnumber% <=0 GOTO Changebutton  
GOSUB Startbutton: WINDOW 1: RETURN





```
Erasebutton:WINDOW 2:CLS
INPUT"Erase which number";E%
CLS:WINDOW 1: BUTTON CLOSE E%
RETURN
```

```
Changeedit:
WINDOW 2: CLS
INPUT"Enter Edit Field number:", enumber%
IF enumber%<=0 GOTO Changeedit
GOSUB Startedit: WINDOW 1: RETURN
Eraseedit:
WINDOW 2: CLS
INPUT"Erase which Edit Field
number";E%: CLS: WINDOW 1
EDIT FIELD CLOSE e%: RETURN
```

```
Quit: WINDOW CLOSE 2:MENU RESET:END
```

```
Startbutton:
CLS:PRINT "Enter Button #";bnumber%;
INPUT" Title: ",Title$
In2:INPUT "Enter Type (1=Push button, 2=Check box,
3= Radio button):",Type
IF Type < 1 OR Type >3 GOTO In2
In3:INPUT "Enter State (0=Inactive, 1=Active/not selected,
2=Active/selected):",State
IF State <0 OR State >2 GOTO In3
CLS:RETURN
```

```
Startedit:
CLS:PRINT "Enter Edit Field #";enumber%;
INPUT" Default (CR=None) : ",default$
e2:INPUT "Enter Type (1=Draw box/no CR, 2=Draw box/CR,
3=No Box/no CR, 4=No Box/CR):",EditType
IF EditType < 1 OR EditType >4 GOTO e2
e3:INPUT "Enter Justify mode (1=Left Justify, 2=Center text,
3=Right Justify):",Justify
IF Justify <1 OR Justify >3 GOTO e3
CLS:RETURN
```

```
window size:
WINDOW OUTPUT 1:winwidth= WINDOW(2)
winheight=WINDOW(3)
WINDOW OUTPUT 2: LOCATE 2,1:PRINT
"Window width=";winwidth;" height=";
winheight;
WINDOW OUTPUT 1:RETURN
```



# Basic Pokes

## Poke The Screen Direct



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No.5

Mike Steiner and Michael M. Boy contributed to this article and we acknowledge their valuable contributions.

### Professor Mac's Screen Pokes

One day, Professor Mac was poking around in his Mac's memory when suddenly he fell in and got stuck. Then along came Mike Steiner with screen memory locations. Professor Mac was then able to write his own program to poke his head out of the Mac screen.

The upper left corner of the screen starts at 108288. The locations progress consecutively starting from the upper left corner of the screen to the bottom right corner moving left to right and down the screen. The bottom right corner of the screen is 130175. These locations are for a 128K Mac. Add  $(512-128)*1024$  for a 512K Mac. Be careful to not poke locations before or after the start and end addresses... unpredictable things will happen.

There are 21888 locations mapping the Macintosh screen. Each line (342 lines) of the screen is represented by 64 memory locations. Thus the first line of the screen goes from 108288 to 108351, the second line starts at 108352 and so on.... (remember to add for 512K Mac)

Each memory location contains 1 sixteen bit word which is bit mapped on the Mac screen. From left to right on the screen, the sixteen bits are mapped from hi order bit to low. A typical memory location is shown in figure 1.

The program pokes zeros into the first 18 lines of the screen, hiding the BASIC menu bars. The menus are still active and can be used to trace or stop the program. The image of Professor Mac is read from data statements and poked into the screen memory locations. Most of the time, you probably won't want to poke the screen directly, there are easier ways to print on the screen. But this method could be used to check status of a particular location of the screen. The FRE(0) command is used to check which Mac (128K or 512K) is being used because the start and ending screen locations are different and have to be adjusted for the 512K Mac.

'Professor Mac's Screen Poke  
'By Dave Kelly

WINDOW CLOSE 1

start=108288! 'Starting address for 128K MAC  
ending=130175! 'Ending address for 128K

mac512=(512-128)\*1024 'address offset  
for 512K MAC

IF FRE(0)>100000! THEN

start=start+mac512:

ending=ending+mac512

erase.top.of.screen:

FOR i= start TO start +64\*18

POKE i,0

NEXT i

WINDOW 1

TEXTFONT(2):TEXTSIZE(18)

LOCATE 8,10:PRINT "Prof. Mac says: Read MacTutor!"

TEXTSIZE(12):PRINT

PRINT TAB(22);"Hit mouse button to quit"

print.professor.mac:

a=start+64\*100 'Start at line 100

b=67 'print 67 lines

FOR j=a TO a+b\*64 STEP 64

FOR k= j+30 TO j+36

READ value

POKE k,value

NEXT k

NEXT j

ON MOUSE GOSUB quit

MOUSE ON:RESTORE

GOTO print.professor.mac

quit:

MENU RESET:END

prof.mac.plot.data:

1 DATA &H00,&H00,&H00,&HE0,&H00,  
&H00,&H00

2 DATA &H00,&H00,&H01,&HFF,&H00,  
&H00,&H00

3 DATA &H00,&H00,&H07,&HFC,&H00,  
&H00,&H00

4 DATA &H00,&H00,&H0F,&HFF,&H00,  
&H00,&H00

5 DATA &H00,&H00,&H3F,&HFF,&HC0,  
&H00,&H00

6 DATA &H00,&H00,&H7F,&HFF,&HE0,  
&H00,&H00

7 DATA &H00,&H01,&HFF,&HFF,&HF8,  
&H00,&H00

◀ Left side of screen

Right side of screen ▶

Prev. location F E D C B A 9 8 7 6 5 4 3 2 1 0 Next location

Typical memory location

Figure 1



8 DATA &H00,&H03,&HFF,&HFF,&HFE,  
&H00,&H00

9 DATA &H00,&H07,&HFF,&HFF,&HFF,  
&H80,&H00

10 DATA &H00,&H1F,&HFF,&HFF,&HFF,  
&HC0,&H00

11 DATA &H00,&H3F,&HFF,&HFF,&HFF,  
&H0F,&H00

12 DATA &H00,&HFF,&HFF,&HFF,&HFF,  
&HFC,&H00

13 DATA &H01,&HFF,&HFF,&HF3,&HFF,  
&HFE,&H00

14 DATA &H03,&HFF,&HFF,&HE1,&HFF,  
&HFF,&H80

15 DATA &H07,&HFF,&HFF,&HE1,&HFF,  
&HFF,&HC0

16 DATA &H0F,&HFF,&HFF,&HF2,&H7F,  
&HFF,&HC0

17 DATA &H1F,&HFF,&HFF,&HFF,&H9F,  
&HFF,&H80

18 DATA &H0F,&HFF,&HFF,&HFF,&HE7,  
&HFF,&H00

19 DATA &H07,&HFF,&HFF,&HFF,&HF9,  
&HFE,&H00

20 DATA &H01,&HFF,&HFF,&HFF,&HFE,  
&H7C,&H00

21 DATA &H03,&HFF,&HFF,&HFF,&HFF,  
&H98,&H00

22 DATA &H03,&H3F,&HFF,&HFF,&HFF,  
&HE8,&H00

23 DATA &H06,&H0F,&HFF,&HFF,&HFF,  
&HFC,&H00

24 DATA &H06,&H07,&HFF,&HFF,&HFF,  
&HFF,&HCC

25 DATA &H06,&H01,&HFF,&HFF,&HFF,  
&H87,&H00

26 DATA &H06,&H00,&H7F,&HFF,&HFF,  
&H0F,&H80

27 DATA &H06,&H00,&H1F,&HFF,&HFF,  
&H0F,&H80

28 DATA &H06,&H00,&H0F,&HFF,&HFF,  
&H0F,&H80

29 DATA &H06,&H03,&H83,&HFF,&HFB,  
&H05,&H00

30 DATA &H06,&H07,&HC0,&HFF,&HF3,  
&H00,&H00

31 DATA &H06,&H0F,&H60,&H3F,&HE3,  
&H0A,&H80

32 DATA &H06,&H0E,&H30,&H1F,&HC3,  
&H05,&H00

33 DATA &H06,&H0C,&H00,&H07,&H83,  
&H02,&H00

34 DATA &H06,&H00,&H00,&H01,&H03,  
&H08,&H80

35 DATA &H06,&H00,&H00,&H00,&H03,  
&H05,&H00

36 DATA &H06,&H00,&H00,&H00,&H03,  
&H00,&H00

37 DATA &H06,&H00,&H00,&H00,&H03,  
&H0A,&H80

38 DATA &H06,&H80,&H70,&H70,&H0B,  
&H05,&H00

39 DATA &H06,&H81,&HF8,&HFC,&H0B,  
&H00,&H00

40 DATA &H06,&HC7,&HFD,&HFF,&H1B,  
&H0A,&H80

41 DATA &H06,&H7F,&HFF,&HFF,&HF3,  
&H00,&H00

42 DATA &H06,&H3F,&HFF,&HFF,&HE3,  
&H1F,&H41

43 DATA &H03,&H07,&HFD,&HFF,&H06,  
&H04,&H63

44 DATA &H03,&H81,&HF8,&HFB,&H0E,  
&H04,&H55

45 DATA &H01,&HC0,&H00,&H00,&H1C,  
&H04,&H49

46 DATA &H00,&HE0,&H00,&H00,&H38,  
&H04,&H41

47 DATA &H00,&H7F,&HFF,&HFF,&HF0,  
&H00,&H00

48 DATA &H00,&H1F,&HFF,&HFF,&HC0,  
&H00,&H00

49 DATA &H00,&H08,&H00,&H00,&H40,  
&H00,&H00

50 DATA &H00,&H08,&H00,&H00,&H40,  
&H00,&H00

51 DATA &H00,&H0F,&HFF,&HFF,&HC0,  
&H00,&H00

52 DATA &H00,&H1A,&HAA,&HAA,&HA0,  
&H00,&H00

53 DATA &H00,&H35,&H55,&H55,&H50,  
&H00,&H00

54 DATA &H00,&H6F,&HEA,&HFE,&HA8,  
&H00,&H00

55 DATA &H00,&HD8,&H35,&H83,&H54,  
&H00,&H00

56 DATA &H01,&HB0,&H1B,&H01,&HAA,  
&H00,&H00

57 DATA &H03,&H50,&H15,&H01,&H55,  
&H00,&H00

58 DATA &H06,&HA8,&H2A,&H82,&HAA,  
&H80,&H00

59 DATA &H0D,&H57,&HD5,&H7D,&H7F,  
&H40,&H00

60 DATA &H1A,&HFA,&HAF,&HAA,&HC1,  
&HA0,&H00

61 DATA &H35,&H05,&H50,&H55,&H80,  
&HD0,&H00

62 DATA &H6A,&H02,&HA0,&H2A,&H80,  
&HA8,&H00

63 DATA &HD6,&H03,&H60,&H35,&H41,  
&H54,&H00

64 DATA &HAB,&H06,&HB0,&H6A,&HBE,  
&HAC,&H00

65 DATA &HD5,&HFD,&H5F,&HD5,&H55,  
&H54,&H00

66 DATA &HAA,&HAA,&HAA,&HAA,&HAA,  
&HAC,&H00

67 DATA &HD5,&H55,&H55,&H55,&H55,  
&H54,&H00

68 DATA &HFF,&HFF,&HFF,&HFF,&HFF,  
&HFC,&H00



# Basic School

## The Art of Clipboarding



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 6

One of the features that makes the Macintosh unique from other computers is the use of the clipboard. The clipboard adds flexibility to the Mac user interface. With it you can exchange data from one application to another. Anyone who has worked with MacPaint or MacWrite is familiar with the operation of the clipboard. What you cut or copy is saved in the clipboard, allowing you to paste it somewhere else. But how do you deal with the clipboard from your own programs? That is the subject we want to explore this month.

Support of the clipboard is another feature of the Macintosh user interface that was left out of MSBASIC version 1.0 but fortunately is easily used in version 2.0. Most of the features that we will use here will only work for version 2.0. There are a few things you should keep in mind when writing programs to use the clipboard.

First, there are three different ways which BASIC can address the clipboard file. Use the **OPEN** statement with one of the following statements for the filename:

"CLIP:" for transferring data from programs that have tabular data, like Multiplan or Chart.

"CLIP:TEXT" for transferring text to and from word processors and other programs.

"CLIP:PICTURE" for transferring picture data to and from MacPaint or other programs.

Files that are **OPENed** to this device, using the mode indicated above, will read/write data directly from/to the Clipboard File stored on the system disk.

The sample program 'Show Clip' shows how the clipboard could be read within an application and displayed. You have probably seen many applications that give the user the option to "Show Clipboard". The program will check to see if the clipboard contains picture data or text and will read the data accordingly and display it in the window. This routine is an example to show how to read from the clipboard, but could be used in your own application to produce the "Show Clipboard" option.

When you make use of **EDIT FIELDS** in your programs, you should keep the Edit menu active so that your **EDIT FIELD** can be fully edited using the cut, copy and paste feature of the Edit menu. This way the clipboard is being used without having to do a whole lot of programming; however the operation is still manual. The user must decide what to select and copy or cut it into the **EDIT FIELD**. The same is true for inserting text with the paste option.

If the file is opened to "CLIP:" or "CLIP:TEXT", the data is read sequentially from the clipboard file. There are actually two clipboard areas. One is the clipboard file and the other is

a temporary area in memory. That's why sometimes the disk turns on when you cut or copy something and other times it doesn't.

To write text to the clipboard:

- 1) Open the clipboard with **OPEN "CLIP:" FOR OUTPUT AS #1** or equivalent.
- 2) Use **Write #1** or **Print #1** to write your variable to the clipboard file.
- 3) **CLOSE #1** to close the file.

To read from the clipboard:

- 1) Open the clipboard with **OPEN "CLIP:" FOR OUTPUT AS #1** or equivalent.
- 2) Use **INPUT #1** to read variables from the clipboard file.
- 3) **CLOSE #1** to close the file.

See the program for an example of this.

You should remember to use the proper format when storing data in the clipboard so that other applications may use the data. For example, the text that a word processor uses would contain format control characters embedded in the text. These kind of characters should only be left in the text stored in the clipboard if you know that the program that will be reading it will use them. Some things might not matter what format they are stored in and the data can be formatted once it has been read into the new application. Most of the applications available will tell you how the data is formatted, so if you know what the data will be used in, there won't be any problem. Some help on transferring files to and from the clipboard and other applications can be found on page 55 of the BASIC manual.

Copying pictures is just about as easy as text. To transfer something to MacPaint:

- 1) Use the **PICTURE ON** statement to record the graphics statements.
- 2) Issue all the graphics statements you need to produce the picture. You don't have to draw it to record it as a picture.
- 3) Use **PICTURE OFF** to stop recording graphics.

(A good example of this is found in the BASIC manual on page 205.)

- 4) Next, open the clipboard file with **OPEN "CLIP:PICTURE" FOR OUTPUT AS #1** (or equivalent statement)
- 5) Send the picture to the clipboard with **PRINT #1,PICTURES\$**
- 6) Close the file: **CLOSE #1**

Next, you can either exit BASIC and paste the picture into MacPaint, or save the picture in the Scrapbook to use later.

To transfer MacPaint pictures to BASIC:

- 1) Put the MacPaint picture in the Clipboard
- 2) Open the clipboard file with `OPEN "CLIP:PICTURE"` `FOR INPUT AS #1` (or equivalent statement)
- 3) Transfer the picture to a string variable (called `image$` in this case): `image$=INPUT$(LOF(1),1)`
- 4) Close the file: `CLOSE #1`
- 5) Draw the Picture to the screen exactly the way it was recorded: `PICTURE,image$`

See the program for an example of reading a picture from the clipboard.

To determine if the data in the clipboard is picture or text we can use the BASIC statement `LOF`. `LOF(1)` will return the length of the specified file (in this case our file is "CLIP:"). If the result turns out to be zero, it means that either the data is a picture or the clipboard is empty. So next, just read whatever picture is stored, if any. If the clipboard is empty, the resulting string variable will also be empty.

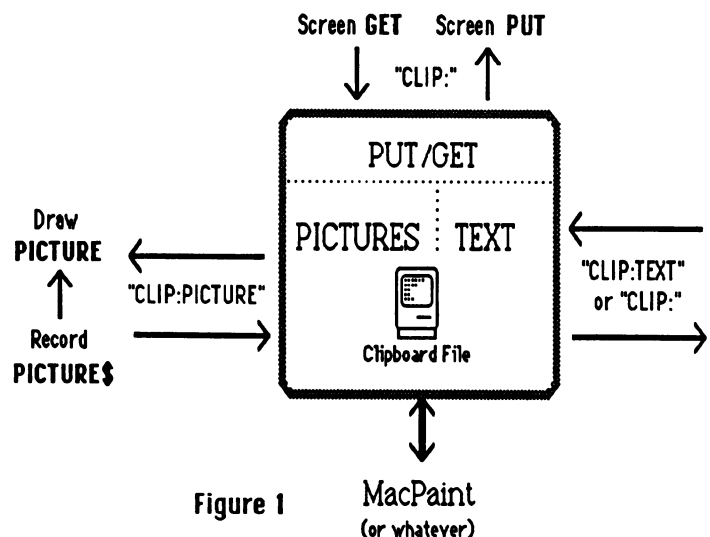
There is one more way to store text or graphics in the clipboard. (I'm not sure that there is a good reason to want to do this, though.) Using a screen `GET`, a portion of the screen can be copied into a non-string variable array. This variable can be written to the clipboard for later use. You can then read the clipboard and use a screen `PUT`, to place the data back on the screen. The screen `GET` and `PUT` commands are much more useful in moving sections of the screen image. Since the array used is not a string, it is not in the proper format for MacPaint or for `PICTURE` statements. I tried to use screen `GET` and `PUT` to read the picture of Professor Mac found in last month's Screen Poke article. The problem is that when the image is directly poked on the screen, BASIC doesn't recognize it as being anything useful. You can't record pokes with the `PICTURE ON/OFF` feature of BASIC. Perhaps one of our readers knows how to change the data from a screen `GET` format to MacPaint picture format. For most programs you will want to use the other methods mentioned anyway.

In conclusion, figure 1 shows the different ways that data can be stored in the clipboard. Most of these methods are explained fairly well in the BASIC 2.0 manual. (ref. pg. 55-58 for more information on transferring data between BASIC and other programs. Note there is an error on page 58 where it says `PICTURES$,IMAGES$` should be `PICTURE,IMAGE$`. Also see pg. 205 for `PICTURE` statement.)

Show Clip  
By Dave Kelly  
©MACTUTOR 1985

```
'Set up windows
WINDOW 1,"Output Window", (2,40)-(510,200),1
WINDOW 2,"Clipboard File", (2,220)-(510,340),1
WINDOW CLOSE 2
'Set up menus
MENU 4,0,0,""
```

```
MENU 5,0,1,"Windows"
MENU 5,1,1,"Output Window"
MENU 5,2,1,"Show Clipboard"
MENU 5,3,1,"Quit"
ON MENU GOSUB handlemenu
MENU ON
ON DIALOG GOSUB handlewindow
DIALOG ON
EDIT FIELD 1,"", (150,42)-(365,120),1
show:GOTO show:GOSUB showclipboard
handlemenu: 'Menu handler
number=MENU(0)
IF number<>5 THEN RETURN
item=MENU(1):MENU
IF item=1 THEN WINDOW 1
IF item=2 THEN GOSUB showclipboard
IF item=3 THEN WINDOW CLOSE 2:
WINDOW CLOSE 1:MENU RESET: END
RETURN
handlewindow: 'Menu handler
stat=DIALOG(0)
IF stat=3 THEN WINDOW DIALOG(3)
IF stat=4 THEN WINDOW CLOSE DIALOG(4)
RETURN
showclipboard:
WINDOW 2:CLS
DIALOG STOP:MENU STOP
'Read text from clipboard
OPEN "CLIP:TEXT" FOR INPUT AS #1
IF LOF(1)=0 THEN CLOSE #1:GOTO do.picture
LOCATE 1,1
WHILE NOT EOF(1)
INPUT #1,a$:PRINT a$
WEND
CLOSE #1:DIALOG ON:MENU ON
RETURN
do.picture: 'Read picture from Clipboard
DIALOG STOP:MENU STOP
OPEN "CLIP:PICTURE" FOR INPUT AS #1
image$=INPUT$(LOF(1),1)
PICTURE,image$
CLOSE #1:DIALOG ON:MENU ON:RETURN
```



# Basic School

## The Standard File Interface From Basic



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 7

A major concern in any program development is that of memory management. This is especially true on the 128K Macintosh. MSBasic leaves only about 14K free program space to work with. However, on the 512K Mac there is over 330K free program space. If you are disappointed by the memory limitations, there are some ways which you may get around some of them.

There are three different areas of memory you can control:

1. The stack
2. The Heap
3. BASIC's data segment

### The Stack and the Heap

Macintosh applications can allocate and release memory by using the stack or the heap. The stack is used to temporarily store information telling BASIC where to return to from GOSUBs, FOR... NEXT loops, WHILE... WEND, subprogram calls, and nested user defined functions. The Macintosh ROM routines require a lot of stack space depending on the number of levels of nesting of controls (such as BUTTONS, EDIT FIELDS). The information on the stack is stored in LIFO (last-in-first-out) order. The last item put on the stack is always the first to be removed. The information is always released at the top of the stack, never in the middle, so there can never be blank "holes" in the stack. As in figure 1, the stack starts at a fixed address in high memory and as information is added to the stack it grows toward low memory (top of stack).

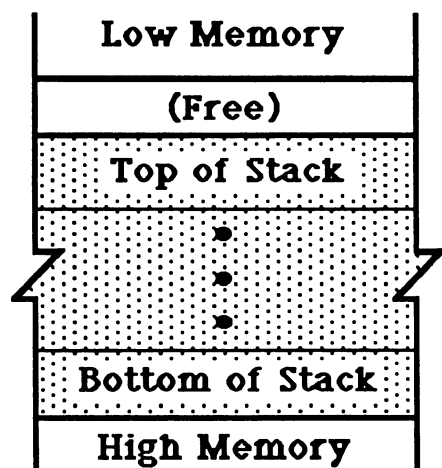


Figure 1

The heap contains blocks of memory which are allocated and released as needed by the Macintosh Operating System's Memory Manager. The Memory Manager keeps track of the heap section of memory and "compacts" the heap if necessary in order to conserve the heap space. If you notice when you load BASIC only a part of BASIC loads at a time. Part of BASIC is in memory, and the rest is in sections that are loaded into memory as needed. Because the heap is smaller on the 128K Mac, there is considerably more disk access involved as sections of BASIC are swapped in and out of the heap. Whenever you use any of the Macintosh "features" such as MENU, BUTTON, EDIT FIELD, PICTURE, SOUND, WAVE, or WINDOW, a part of the heap is used to keep track of these "resources". Open desk accessories also use a portion of the heap. The heap starts at low memory and adds blocks toward high memory. After a program has been running for awhile the heap will become fragmented with "holes" in the middle as various resources are released. By closing WINDOWS, MENUS and closing desk applications memory can be made available in the stack for other resources.

The SOUND/WAVE buffer can be released (1024 bytes of heap) by using a WAVE 0 statement when it is no longer needed. Also a PICTURE ON immediately followed by PICTURE OFF will reclaim memory which was used by a previous picture that was in the heap. The Memory Manager will compact the contents of the heap.

One note on the speed of your programs as it pertains to the heap. I have found that when first defining EDIT FIELD and BUTTON controls that it takes longer the first time through the program while the heap allocates memory to keep the BUTTON and EDIT FIELD resources. Re-opening a window that had been closed which contains buttons and edit fields will use the same resources which were previously allocated in the heap. There are always two heap areas in memory: the system heap, which is used by the Toolbox and Operating System, and the application heap, which is used by the application program (in our case the application is MSBASIC).

The BASIC data segment area of memory is the area used to store the BASIC program and variables. This area also contains area for file buffers for opened files.

### CLEAR AND FRE(n)

MSBASIC has provided the CLEAR statement to allocate memory to the three areas of RAM mentioned above. The CLEAR statement adjusts the number of bytes reserved for the stack and the data segment. The syntax is:

**CLEAR** [, *data-segment-size* ], *stack-size* ]]

The remaining RAM is left for the heap space. The heap space is calculated by taking the total amount of RAM (128K or 512K) minus the data segment size and the stack size (*heap* = Total RAM - ( *data-segment-size* + *stack-size* )). For most programs on a 512K Mac it is not necessary to use the CLEAR statement, but it is needed for many programs to run on the 128K Mac. You should keep this in mind if you intend for your programs to run on any Mac (128K or 512K). You can use the statement **FRE(*n*)** to find out how much free memory is available in each part of RAM. **FRE(-1)** returns the amount of free memory in the heap. **FRE(-2)** returns the amount of stack which has **never** been used. By using this value the program can be adjusted using **CLEAR** to use memory the most efficiently. Be sure that the worst case is used when fine tuning the memory. If (*n*) is (" ") or any other number (except -1 or -2) the expression returns the number of free bytes available in BASIC's data segment. All of the **FRE** statements will compact string space. Each time a string is defined in BASIC, part of the data segment area of memory is used. After swapping and shuffling strings around in your program the data segment becomes full of strings, most of which are no longer needed. For example if a string **A\$** is assigned as "MACINTOSH" and then reassigned as "MACTUTOR", new space is allotted for "MACTUTOR" but the old string still exists in memory. **A\$** would only point to the most recent assignment of **A\$**. By using the **FRE(*n*)** statement, garbage collection is done and all the currently assigned strings are compacted in the data segment of memory.

### Using Run or Chain

If you find that using the **CLEAR** statement still doesn't give you enough memory you can split your program into subprograms and load in each program as needed. The **RUN** statement may be used to load and execute another BASIC program. The syntax is **RUN filename[,R]**. The **R** added to the end of the statement will cause all open data files to remain open. The problem with using the **RUN** statements is that when **RUN** executes, all variables are erased. This means that all variables which are needed in the next program segment would need to be saved (temporarily) to the disk. **RUN** is best used to load new programs that are independent from the calling program.

To preserve your variables you should use the **CHAIN** statement. The syntax is: **CHAIN [ MERGE [*filespec* [,*expression*] [,ALL] [, DELETE *range*] ]]**. The **MERGE** statement appends the called program to the end of the program currently in memory. The called program must have been saved as an ASCII file for this to work. *filespec* is the specification (disk name and filename) and *expression* is an expression or line number which tells BASIC where to start executing the called program. An alphanumeric label may not be used as this expression. If you use the **ALL** option, all the variables in the current program in memory are passed to the called program. The **DELETE** statement is used to delete lines of program currently in memory to make

room for the called program. The BASIC manual is clear describing the syntax used here.

### Choosing Your Filename

Some of these statements may seem to be somewhat trivial, but it takes some planning to determine when and how each section of program should be loaded and executed. The program **FILE\$ Demo** demonstrates a way that the user can select which program or file to use. The function **FILE\$(*n* [,*prompt-string*])** is supplied by **MSBASIC** to allow various types of files to be selected using the mini-finder dialog box. I'm sure that we are familiar with it from using it in most applications. The parameter *n* is a number 0 or 1. **FILE\$(0)** calls a dialog box which prompts the user for the name of a file. The *prompt-string* is displayed as the default filename. This is useful when you want to let the user decide which filename to use to save data to the disk. **FILE\$(1)** calls up the mini-finder dialog box and prompts the user to select a filename in the list. You can use this to select files in either disk drive or on other disks. Both commands return the name of the file selected. This filename can then be used to load or save data or to call a new program. The *prompt-string* in the **FILE\$(1)** statement contains a list of the file types, 4 characters per type. The file type is attached to the filename in the directory and the finder uses it to know what kind of file each icon represents.

In the demo program a window opens and asks for the user to select the type or types that should be included in the file selection. This way certain types of files could be screened out. The basic stores files as **TEXT**, but the type may be changed by using the **NAME** statement. In this way you could use a special type that "belongs" only to your program. If no types are selected in the demo program, then the *prompt-string* is blank and all types are selected. After the desired type buttons are selected selecting the "Select Files" button will execute the **FILE\$(1)** statement and the mini-finder dialog box will appear. Select a filename and push the open button and you can see the format that is returned by the **FILE\$(1)** statement. There are many possible ways that this could be used which are left to you to decide for your particular programming application.

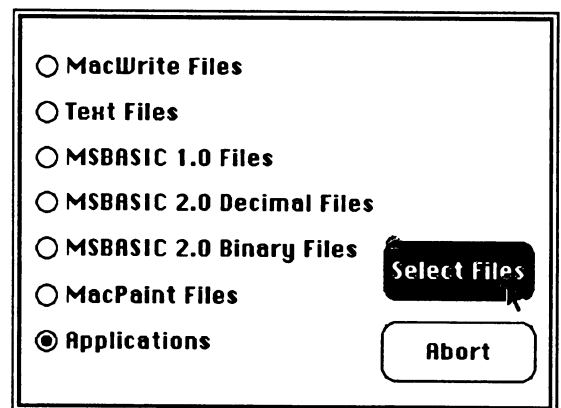


Fig. 2 Selection of File Types

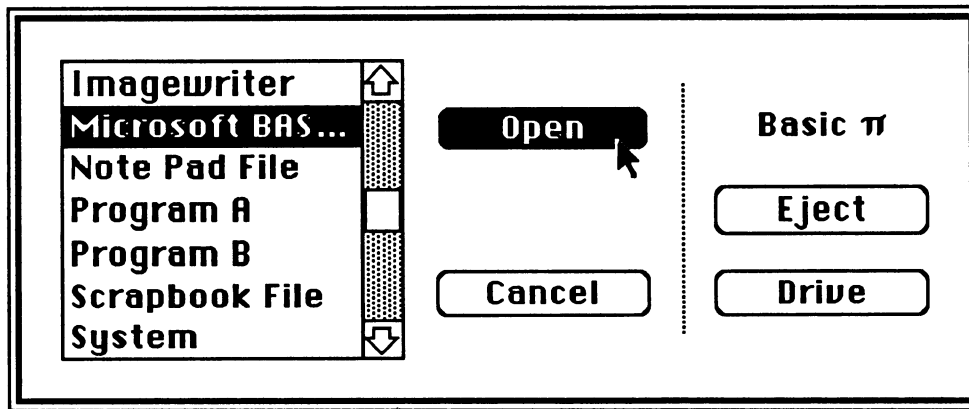


**Fig. 3 Standard File Interface**

FILE\$( ) Demo  
by David Kelly  
©MACTUTOR 1985

```
Erase menus
FOR i=1 TO 5
 MENU i,0,0,""
NEXT
WINDOW 1,,(15,40)-(495,245),2
```

' initialize button status flags

```
FOR i=1 TO 7
 btn(i)=1
```

```
NEXT
```

Begin: 'Set up button controls

```
BUTTON 1,btn(1),"MacWrite Files",
 (119,15)-(237,30),3
```

```
BUTTON 2,btn(2),"Text Files",
 (119,40)-(200,55),3
```

```
BUTTON 3,btn(3),"MSBASIC 1.0 Files",
 (119,65)-(252,80),3
```

```
BUTTON 4,btn(4),"MSBASIC 2.0 Decimal
Files",(119,90)-(307,105),3
```

```
BUTTON 5,btn(5),"MSBASIC 2.0 Binary
Files",(119,115)-(317,130),3
```

```
BUTTON 6,btn(6),"MacPaint Files",
 (119,140)-(233,155),3
```

```
BUTTON 7,btn(7),"Applications",
 (119,165)-(219,180),3
```

```
BUTTON 8,1,"Select Files",
 (363,87)-(448,121)
```

```
BUTTON 9,1,"Abort",
 (363,131)-(448,165)
```

' Wait for button push

```
loop:WHILE Dialog(0) <> 1:WEND
```

```
 buttonpushed=Dialog(1)
```

```
 IF buttonpushed=8 THEN seefiles
```

```
 IF buttonpushed=9 THEN WINDOW
```

```
 CLOSE 1:MENU RESET:END
```

```
 IF btn(buttonpushed)=1 THEN
```

```
 btn(buttonpushed)=2 ELSE
```

```
 btn(buttonpushed)=1
```

```
 BUTTON buttonpushed,
```

```
 btn(buttonpushed)
```

```
GOTO loop
```

```
seefiles:
```

- ' MSBA Basic 1.0
- ' MSBB Basic 2.0 Decimal version
- ' MSBC Basic 2.0 Binary version
- ' TEXT Text file
- ' APPL Application
- ' PNTG MacPaint File
- ' WORD MacWrite File
- ' MPRJ MacProject File
- ' Other types may be added

```
type$=""
```

```
IF btn(1)=2 THEN type$=type$+"WORD"
```

```
IF btn(2)=2 THEN type$=type$+"TEXT"
```

```
IF btn(3)=2 THEN type$=type$+"MSBA"
```

```
IF btn(4)=2 THEN type$=type$+"MSBB"
```

```
IF btn(5)=2 THEN type$=type$+"MSBC"
```

```
IF btn(6)=2 THEN type$=type$+"PNTG"
```

```
IF btn(7)=2 THEN type$=type$+"APPL"
```

```
selection$=FILE$(1,type$)
```

```
IF selection$="" THEN loop
```

' Close buttons

```
FOR i=1 TO 9
```

```
 BUTTON CLOSE i
```

```
NEXT
```

```
PRINT "You have selected ";
```

```
CALL TEXTFACE(1)
```

```
PRINT selection$
```

```
CALL TEXTFACE(0)
```

```
PRINT "Types =",type$
```

```
BUTTON 1,1,"More", (363,87)-(448,121)
```

```
BUTTON 2,1,"Abort",
 (363,131)-(448,165)
```

' Wait for button push

```
WHILE Dialog(0) <> 1:WEND
```

```
 buttonpushed=Dialog(1)
```

```
 IF buttonpushed=1 THEN CLS
```

```
 GOTO Begin
```

```
 WINDOW CLOSE 1:MENU RESET:END
```



# Basic School

## Printing Techniques



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 8

This month's column features a guide to printing on the Macintosh. The ability of the Macintosh to make our documents look good is one of the things that attracted me to the Mac. There are some choices to make when printing via Microsoft BASIC.

### MSBasic Printing

MSBASIC offers the following commands for printing:  
**BASIC PRINT COMMANDS**

- LCOPY** - Sends a copy of screen to the printer just like command shift-4. (standard quality)
- LIST filename** - see below for valid filenames.
- LLIST** - same as **LIST filename** when filename = "LPT1:DIRECT" see more below.
- LPOS** - gives the position of the print head (not necessarily the physical position).
- LPRINT** - works just like **PRINT** except it prints to the printer instead of the screen. Draft quality.
- LPRINT USING** - works just like **PRINT USING** except it prints to the printer instead of the screen. Draft quality
- OPEN # / CLOSE #** - output to printer is specified by the *filespec*. Used the same as in storing files to a sequential file.
- PRINT #** - use with **OPEN** and **CLOSE**. Same as output to a sequential file.
- PRINT # USING** - use with **OPEN** and **CLOSE**. Same as output to a sequential file.
- WIDTH #** - used to set the line width *size* and *print-zone* parameters (similar to tab stops) for the printer.
- WIDTH output-device** - same as **WIDTH #**
- WIDTH LPRINT** - same as **WIDTH #**
- WINDOW OUTPUT #** - allows the printer to print subsequent graphics as determined by subsequent graphics statements such as **CIRCLE**, **PSET**, **PICTURE**, and calls to the Macintosh ROM routines. See sample program.
- WRITE #** - use with **OPEN** and **CLOSE**. Same as output to a sequential file.

The *filespec*, *output-device*, or *filename* may be one of the following:

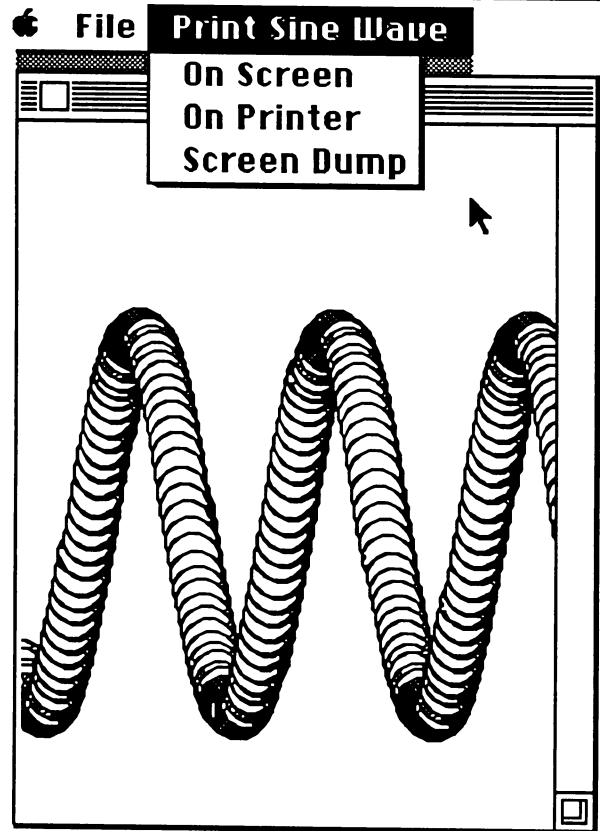


Fig. 1: Screen Output

"LPT1:" - All data written to a file opened to this *output-device* will be directed to the lineprinter. Prints in standard quality and issues a printer form feed when done. Supports graphics.

"LPT1:DIRECT" - When this is used, BASIC sends data to the printer as ASCII characters. The printer will respond to all normal printer codes just as it would connected to some other computer. Prints draft quality with no form feed. Fastest way to print. No graphics.

"LPT1:PROMPT" - This calls the standard page setup and printer quality dialog boxes and allows you to choose and quality of printing. Also allows 50% reduction if you have the new imagewriter driver installed. There is no known way to print high quality without using this *output-device*. Perhaps there is a way to call the imagewriter driver directly?? Graphics is supported only when High and Standard quality print is selected.

Now the decision is up to you. MSBASIC supplies the tools to print just about anything that can be printed to the screen. When the *file-spec* is set to one of the modes that supports graphics, (LPT1: or LPT1:PROMPT) whatever font has been specified with the TEXTFONT and other related commands will appear in the proper font as expected. When printing in draft quality modes, graphics is not supported and the font used is the one built in to the printer. One thing missing is the ability to output high quality printing without having to go through the dialogs. There may be a way to call the imagewriter driver directly to do this, but I haven't discovered how to do that yet.

It should be noted that the TAB and SPC functions may only be used with PRINT and LPRINT. This should not be much of a problem though.

The program below demonstrates how to print graphics created within a MSBASIC program. The "LPT1:PROMPT" *output-device* specification is used. This allows you to print with any quality of printing. The program uses the PICTURE statement to create a sine wave drawing. The menu lets you print on the screen or the printer or do a screen dump using the LCOPY statement. The WINDOW OUTPUT # statement is used to send graphics output to the printer. The WINDOW OUTPUT # statement saves graphics commands and sends them to the printer when the output file is closed. In this program I have used the ROM functions PAINTOVAL, ERASEOVAL, and FRAMEOVAL to draw a sinewave either on the screen or on the printer. The ROM functions are well documented in the MSBASIC 2.0 manual.

NOTE THAT THIS ROUTINE DOES NOT WORK WITH THE LASER PRINTER. MACTUTOR WILL PAY \$50 TO THE FIRST ARTICLE SUBMISSION THAT EITHER MAKES THIS PROGRAM WORK ON THE LASER OR GIVES A WELL-DOCUMENTED ACCOUNT OF WHY IT DOESN'T.

### MacBasic Printing

The Macintosh BASIC Handbook, by Thomas Blackadar and Jonathan Kamin makes no reference printing graphics on the printer. Apparently Apple has spent a lot of time trying to develop MacBASIC as an "educational" tool and hasn't developed all the features to make it into a serious development language. According to this book, and few other "pre-mature" books that are out, the TOOLBOX function will not even be documented in the initial release of MacBASIC. The TOOLBOX includes the means of creating windows, menus, and other Mac controls. Printing is limited to draft quality via a file aparameter named .PRINTER. The fancy features of MacBASIC only apply to the Mac screen.

' Sine Wave  
' by Dave Kelly  
' ©MACTUTOR 1985

```
MENU 1,0,1,"File"
MENU 1,1,1,"Quit"
MENU 5,0,0,""
MENU 4,0,0,""
MENU 3,0,0,""
MENU 2,0,0,""
ON MENU GOSUB premenu: MENU ON
WINDOW OUTPUT 1
period=70: amplitude=70
phase=0: offset=120
pi=3.141592654#: w=(2*pi)/period
CLS:PRINT" Please wait while picture data
 is being recorded (approx 25 sec.)"
```

```
PICTURE ON
FOR x=10 TO 525
```

```
' Sine wave equation
y=offset+amplitude*SIN((w*x)+phase)
```

```
'Set up parameters for PAINTOVAL
rectangle%(0)=y-19: rectangle%(1)=x-19
rectangle%(2)=y: rectangle%(3)=x
PAINTOVAL(VARPTR(rectangle%(0)))
```

```
'Setup parameters for ERASEOVAL
rectangle%(2)=y-2: rectangle%(3)=x-2
ERASEOVAL(VARPTR(rectangle%(0)))
```

```
'Setup parameters for FRAMEOVAL
rectangle%(0)=y-20: rectangle%(1)=x-20
rectangle%(2)=y-1: rectangle%(3)=x-1
FRAMEOVAL(VARPTR(rectangle%(0)))
NEXT x
PICTURE OFF:MENU OFF
```

```
'Setup menus
MENU 2,0,1,"Print Sine Wave"
MENU 2,1,1,"On Screen"
MENU 2,2,1,"On Printer"
MENU 2,3,1,"Screen Dump"
```

```
ON MENU GOSUB menuselect:MENU ON
CLS: PRINT"Make your ^^^^^^^^^^
 selection from this menu"
loop: GOTO loop
```

```
premenu:
menunumber=MENU(0):MENU
IF menunumber=1 THEN PICTURE
 OFF: GOTO quit
RETURN
```

```
menuselect:
menunumber=MENU(0):MENU
IF menunumber =1 THEN quit
IF menunumber<>2 THEN RETURN
menuitem=MENU(1)
```



```
'Print graphics on Screen
IF menuitem=1 THEN CLS:GOSUB
 printpic:RETURN
```

```
'Screen dump
' Note that LCOPY does not work with
' laser printers. Anybody know why?
IF menuitem=3 THEN CLS:GOSUB
 printpic:LCOPY:RETURN
```

```
'Print graphics on printer
' Note that this does not work with laser
' printers. Anybody know why?
OPEN "LPT1:PROMPT" FOR OUTPUT AS #1
```

#### WINDOW OUTPUT #1

```
GOSUB printpic
CLOSE #1
RETURN
```

```
printpic:
MENU 1,0,0:MENU 2,0,0
PICTURE(2,40)-(510,340),PICTURES$
MENU 1,0,1:MENU 2,0,1
RETURN
```

```
quit:
CLS:MENU RESET:END
```



# Special Projects

## The Structure of a Microsoft BASIC Program



Mike Steiner  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 8

### Unraveling the Mysteries

During a fit of boredom and out of curiosity, I started peeking with fEdit at Microsoft BASIC to see how it was formatted when saved in compressed mode.

I found that if a character with the high bit set is not preceded by a quote, REM (or a single quote) or DATA, then the BASIC interpreter considers it a **keyword**, part of the coding of a number, or a syntax error. I received some help from Michael M. Boy of Elgin AZ, who wrote a utility that locates itself in memory (Listing 1). (Programs as presented here are for a 512K Macintosh. Change values as needed for a 128K Mac. Some experimentation may be necessary to find the correct values. When you have found the starting point, you can make the proper adjustments in listings 2 and 3.)

With this utility, we determined that a BASIC program usually starts at **location 77002 (decimal)** on a 512K Macintosh; however, on some occasions programs loaded four to six bytes lower in memory. Apparently, this situation happens when another application is run before BASIC is loaded. If the computer is reset, programs load at 77002. Mike then wrote a routine, which I modified (See listing 2), that prints a hex and ASCII dump of itself. The routine peeks memory from the start of the program until it finds the end of program marker. The memory dump is identical to that stored on disk with the exception that one byte is prefixed to the disk file to show the nature of the program (compressed or protected) and whether it was written in the Binary or Decimal version of BASIC. This, however, is part of another column. Following is a discussion of the format of Microsoft BASIC programs.

### Format of Basic Program Lines

If a program line has *n* bytes, the line format is as follows:

**Bytes 1 and 2:** If the line is not numbered, the first byte of the line has the high bit cleared. If the line is numbered, then this bit is set. The first two bytes (high order byte first) show the length of the line; however only the second digit of the first byte is used. The maximum number of bytes in the line is normally 255. However, if there are colons or REM statements automatically inserted by BASIC (see below for a discussion of tokens automatically inserted by BASIC), the maximum number may exceed 255; the longest line I have seen

had 259 bytes. The line length includes all bytes in the line, including those used internally by BASIC and not displayed in the program listing, such as the end of line marker.

Byte 3 is always \$00.

**Bytes 4 and 5:** If the line is numbered, these bytes show the line number, high byte first; the highest line number is 65529. If there is no line number the body of the line starts with byte 4.

**Bytes 6 (4 if no line number) through *n*-1:** This is the information you typed in the line.

**Byte *n*:** Always \$00 to show end of line. This value (\$00) may appear within a line, but if it is not at position *n*, which is coded by the first two bytes, the program recognizes that it is not the end of line marker.

A blank line is represented by "00 04 00 00." This includes the end of line marker. A blank numbered line is shown by "80 06 00 HB LB 00" where HB and LB are the high and low bytes of the line number.

The end of program marker is "00 00 00 00 00" including the end of line marker for the last line of the program. These five zeroes clearly describes the end of the program when the first byte of the sequence is byte *n*. This sequence may also appear in the body of a line as part of the coding of a declared double precision number.

### Data Format Within a Line

All text within quotes or following a REM or DATA statement are represented in positive ASCII (i.e. high bit off). However, those characters that are typed in conjunction with the Option key (e.g. "π" "+" etc.) use negative ASCII (i.e. high bit set). Numbers are coded in positive and negative ASCII. The formatting of numbers is quite complex and is beyond the scope of this column.

Reserved words are represented by negative ASCII. There are only 128 negative ASCII bytes possible, and there are over 200 reserved words; therefore some reserved words are represented by pairs of bytes (both in negative ASCII). (See tables 1 and 2.) Any byte with high bit set that is not defined

as a reserved word or part of the coding for a number and is not part of a PRINT statement, a REM, or a DATA statement is not displayed in the listing and will cause an error message when program execution reaches it.

There are a few special cases: REM, ELSE, GOTO, and GOSUB.

### Special Cases

**REM:** Microsoft BASIC lets you use the apostrophe character as an abbreviation for REM. If you do, it inserts a \$3A (colon) and an \$AF (REM) before the apostrophe (\$E8) token, so what is actually represented is ":REM". When BASIC sees these three bytes, it suppresses listing the ":REM" (\$3A \$AF) in the list window. So, you use one extra bytes of memory whenever you use an apostrophe instead of REM at the beginning of a line (If you use REM, you need to put a space after it; with the apostrophe you do not.) Using it within the line does not use any extra bytes because if you type REM there, you have to precede it with a colon.

**ELSE:** Similarly, if you type ELSE in an IF - THEN statement, BASIC precedes it with a non-printing colon if you do not type one. You do not use any extra bytes in this case because your only other option is to type the colon yourself. You decide whether the colon is visible in the program listing by typing it, or not visible by letting BASIC insert it.

**GOTO (\$97) and GOSUB (\$96)** are followed by "20 1B 00 00" and the label name, if going to a labeled line. If going to a numbered line, the token is followed by "20 0E 00" and the line number, which is represented by two bytes, high byte first.

### Managing Memory

From the above information, we can see that if available memory is a constraint, you are better off using line numbers rather than line labels in your programs. Line numbers use only two bytes in the line whereas a label uses one byte for each character in the label plus one more for the mandatory colon. Further, each reference to a labeled line elsewhere in the program uses five bytes plus the length of the label, whereas a reference to a numbered line always uses exactly five bytes. Of course, if the line is not referenced anywhere in the program, neither a label nor a line number is needed.

### Description of the Goodies

Listing 1 is the locator program that finds itself in memory by searching for the REM token in the first line.

Listing 2 is the poke program that will poke the token of your choice into memory to replace a REM statement, thereby self-modifying the program. This is great for getting mathematical input and then executing it to return the value of

an inputted function. This same technique was used several years ago on the Apple II by several companies to produce plot packages that could take an inputted function string and plot the results. With this utility, you can accomplish this same technique on the Macintosh.

Listing 3 is a program fragment that will do a memory dump of your program in hex and ascii.

Table 1 is a listing of the reserved words in Microsoft Basic, sorted by ASCII code. Use this table with the poke utility to convert remark statements into new BASIC code dynamically.

#### Basic Listing #1 Locator Program

```
REM }}{here
x$ = "}}{here" : REM x$ must be the same as the REM on the
 above line
y$ = LEFT$(x$,1)
x = 42000! : REM start searching here, should be suitable for
 128K Mac at this location
FOR i = x TO 512*1024
 z$ = CHR$(PEEK(i))
 IF z$ <> y$ THEN elp1
 a$ = ""
 FOR j = 1 TO LEN(x$)-1
 a$ = a$ + CHR$(PEEK(i+j))
 NEXT j
 IF a$ = RIGHT$(x$,LEN(x$)-1) THEN PRINT "we got it at ";
 i : END
elp1:
IF i = x THEN PRINT "now at ";x : x = i + 1000
NEXT i
```

**REM** This program does not give the start of the program. It gives the location where the first character in the REM statement begins. Start of program is lower in memory.

#### Basic Listing #2 Poke Token in Memory

```
SUB printit STATIC
SHARED b
PRINT | (b) :REM the vertical bar is a place holder for the
 value to be poked and is replaced with the token for the
 function to be executed by POKE 77031. Run the program
 and list it again. The vertical bar will be replaced by the
 function you selected. DO NOT INSERT ANY TEXT
 BEFORE THE VERTICAL BAR or the program will not
 work. The bar, however, may be replaced by any
 character.
END SUB

OPTION BASE 1
DIM funct (5),funct$(5)

DATA 130, 160, 181,183, 186, ATN, COS, SIN, SQR, TAN

FOR i = 1 TO 5: READ funct (i): NEXT
```

```
FOR i = 1 TO 5: READ funct$(i):NEXT
```

```
CLS
PRINT "Enter Function you want evaluated"
PRINT
PRINT " 1) ATN 2) COS 3) SIN"
PRINT " 4) SQR 5) TAN"
```

```
getfunction: INPUT "Your choice > ",a: IF a<1 OR a>5
 THEN getfunction
INPUT "Enter value to be processed > ",b
POKE 77031!,funct (a):REM Poke the token into memory
```

```
PRINT: PRINT
PRINT "The "; funct$(a); " of "; b; "is ";
CALL printit
```

### Basic Listing #3 Print Memory

```
SUB prtmem STATIC :REM Merge this routine to your
```

program. Then CALL prtmem from the command window.

```
CLS
CALL TEXTFONT (4)
CALL TEXTSIZE (9)
i= 77001! :REM start of program minus 1
WHILE (PEEK(i) + PEEK(i-1) + PEEK(i-2) + PEEK(i-3) +
 PEEK (i-4)) <> 0 : REM Look for end of program marker
 NOTE this may fail if numbers are declared as double
 precision.
 k = k + 1:i = i + 1
 IF k= 1 THEN PRINT USING "#####"; i;"-";
 PRINT RIGHT$("0" + HEX$(PEEK(i)),2);" ";
 IF PEEK(i) > 31 THEN b$ = CHR$ (PEEK(i)): ELSE b$ = "."
 a$ = a$ +b$
 IF k = 8 THEN PRINT TAB (35);a$: k = 0 : a$ = "": REM
 Print 8 bytes then Print ASCII representation
WEND
IF k<> 0 THEN PRINT TAB (35);a$:REM Print ASCII for last
line
END SUB
```

### BASIC TOKEN LIST BY ASCII NUMBER MICROSOFT BASIC 2.0

| <u>TOKEN</u> | <u>DECIMAL</u> | <u>HEX</u> | <u>TOKEN</u>     | <u>DECIMAL</u> | <u>HEX</u> |
|--------------|----------------|------------|------------------|----------------|------------|
| ABS          | 128            | 80         | LOF              | 162            | A2         |
| ASC          | 129            | 81         | LOG              | 163            | A3         |
| ATN          | 130            | 82         | LSET             | 164            | A4         |
| CALL         | 131            | 83         | MID\$            | 165            | A5         |
| CDBL         | 132            | 84         | MKD\$            | 166            | A6         |
| CHR\$        | 133            | 85         | MKI\$            | 167            | A7         |
| CINT         | 134            | 86         | MKS\$            | 168            | A8         |
| CLOSE        | 135            | 87         | NEXT             | 169            | A9         |
| COMMON       | 136            | 88         | ON               | 170            | AA         |
| COS          | 137            | 89         | OPEN             | 171            | AB         |
| CVD          | 138            | 8A         | PRINT            | 172            | AC         |
| CVI          | 139            | 8B         | PUT              | 173            | AD         |
| CVS          | 140            | 8C         | READ             | 174            | AE         |
| DATA         | 141            | 8D         | REM              | 175            | AF         |
| ELSE         | 142            | 8E         | RETURN           | 176            | B0         |
| EOF          | 143            | 8F         | RIGHT\$          | 177            | B1         |
| EXP          | 144            | 90         | RND              | 178            | B2         |
| FIELD        | 145            | 91         | RSET             | 179            | B3         |
| FIX          | 146            | 92         | SGN              | 180            | B4         |
| FN           | 147            | 93         | SIN              | 181            | B5         |
| FOR          | 148            | 94         | SPACE\$          | 182            | B6         |
| GET          | 149            | 95         | SQR              | 183            | B7         |
| GOSUB        | 150            | 96         | STR\$            | 184            | B8         |
| GOTO         | 151            | 97         | STRING\$         | 185            | B9         |
| IF           | 152            | 98         | TAN              | 186            | BA         |
| INKEY\$      | 153            | 99         | VAL              | 188            | BC         |
| INPUT        | 154            | 9A         | WEND             | 189            | BD         |
| INT          | 155            | 9B         | WHILE            | 190            | BE         |
| LEFT\$       | 156            | 9C         | WRITE            | 191            | BF         |
| LEN          | 157            | 9D         | STATIC           | 227            | E3         |
| LET          | 158            | 9E         | USING            | 228            | E4         |
| LINE         | 159            | 9F         | TO               | 229            | E5         |
| LOC          | 161            | A1         | THEN             | 230            | E6         |
|              |                |            | NOT              | 231            | E7         |
|              |                |            | ' (SINGLE QUOTE) | 232            | E8         |
|              |                |            | >                | 233            | E9         |
|              |                |            | =                | 234            | EA         |

| <u>TOKEN</u> | <u>DECIMAL</u> | <u>HEX</u> |
|--------------|----------------|------------|
| <            | 235            | EB         |
| + (PLUS)     | 236            | EC         |
| - (MINUS)    | 237            | ED         |
| *            | 238            | EE         |
| /            | 239            | EF         |
| ^ (CARET)    | 240            | F0         |
| AND          | 241            | F1         |
| OR           | 242            | F2         |
| XOR          | 243            | F3         |
| EQV          | 244            | F4         |
| IMP          | 245            | F5         |
| MOD          | 246            | F6         |
| /            | 247            | F7         |
| AUTO         | 248 128        | F8 80      |
| CHAIN        | 248 129        | F8 81      |
| CLEAR        | 248 130        | F8 82      |
| CLS          | 248 131        | F8 83      |
| CONT         | 248 132        | F8 84      |
| CSNG         | 248 133        | F8 85      |
| DATE\$       | 248 134        | F8 86      |
| DEFINT       | 248 135        | F8 87      |
| DEFSNG       | 248 136        | F8 88      |
| DEFDBL       | 248 137        | F8 89      |
| DEFSTR       | 248 138        | F8 8A      |
| DEF          | 248 139        | F8 8B      |
| DELETE       | 248 140        | F8 8C      |
| DIM          | 248 141        | F8 8D      |
| EDIT         | 248 142        | F8 8E      |
| END          | 248 143        | F8 8F      |
| ERASE        | 248 144        | F8 90      |
| ERL          | 248 145        | F8 91      |
| ERROR        | 248 146        | F8 92      |
| ERR          | 248 147        | F8 93      |
| FILES        | 248 148        | F8 94      |
| FRE          | 248 149        | F8 95      |
| HEX\$        | 248 150        | F8 96      |
| INSTR        | 248 151        | F8 97      |
| KILL         | 248 152        | F8 98      |
| LIST         | 248 153        | F8 99      |
| LLIST        | 248 154        | F8 9A      |
| LOAD         | 248 155        | F8 9B      |
| LPOS         | 248 156        | F8 9C      |
| LPRINT       | 248 157        | F8 9D      |
| MERGE        | 248 158        | F8 9E      |
| NAME         | 248 159        | F8 9F      |
| NEW          | 248 160        | F8 A0      |
| OCT\$        | 248 161        | F8 A1      |
| OPTION       | 248 162        | F8 A2      |
| PEEK         | 248 163        | F8 A3      |
| POKE         | 248 164        | F8 A4      |
| POS          | 248 165        | F8 A5      |
| RANDOMIZE    | 248 166        | F8 A6      |
| RENUM        | 248 167        | F8 A7      |
| RESTORE      | 248 168        | F8 A8      |
| RESUME       | 248 169        | F8 A9      |
| RUN          | 248 170        | F8 AA      |
| SAVE         | 248 171        | F8 AB      |
| STOP         | 248 173        | F8 AD      |
| SWAP         | 248 174        | F8 AE      |
| SYSTEM       | 248 175        | F8 AF      |

| <u>TOKEN</u>  | <u>DECIMAL</u> | <u>HEX</u> |
|---------------|----------------|------------|
| TIME          | 248 176        | F8 B0      |
| TRON          | 248 177        | F8 B1      |
| TROFF         | 248 178        | F8 B2      |
| VARPTR        | 248 179        | F8 B3      |
| WIDTH         | 248 180        | F8 B4      |
| BEEP          | 248 181        | F8 B5      |
| CIRCLE        | 248 182        | F8 B6      |
| LCOPY         | 248 183        | F8 B7      |
| MOUSE         | 248 184        | F8 B8      |
| POINT         | 248 185        | F8 B9      |
| PRESET        | 248 186        | F8 BA      |
| PSET          | 248 187        | F8 BB      |
| RESET         | 248 188        | F8 BC      |
| TIMER         | 248 189        | F8 BD      |
| SUB           | 248 190        | F8 BE      |
| EXIT          | 248 191        | F8 BF      |
| SOUND         | 248 192        | F8 C0      |
| BUTTON        | 248 193        | F8 C1      |
| MENU          | 248 194        | F8 C2      |
| WINDOW        | 248 195        | F8 C3      |
| DIALOG        | 248 196        | F8 C4      |
| LOCATE        | 248 197        | F8 C5      |
| CSRLIN        | 248 198        | F8 C6      |
| LBOUND        | 248 199        | F8 C7      |
| UBOUND        | 248 200        | F8 C8      |
| SHARE         | 248 201        | F8 C9      |
| UCASE\$       | 248 202        | F8 CA      |
| SCROLL        | 248 203        | F8 CB      |
| LIBRARY       | 248 204        | F8 CC      |
| CVSBCD        | 248 205        | F8 CD      |
| CVDBCD        | 248 206        | F8 CE      |
| MKSBCD\$      | 248 207        | F8 CF      |
| MKDBCD\$      | 248 208        | F8 D0      |
| OFF           | 249 244        | F9 F4      |
| BREAK         | 249 245        | F9 F5      |
| WAIT          | 249 246        | F9 F6      |
| USR           | 249 247        | F9 F7      |
| TAB           | 249 248        | F9 F8      |
| STEP          | 249 249        | F9 F9      |
| SPC           | 249 250        | F9 FA      |
| OUTPUT        | 249 251        | F9 FB      |
| BASE          | 249 252        | F9 FC      |
| AS            | 249 253        | F9 FD      |
| APPEND        | 249 254        | F9 FE      |
| ALL           | 249 255        | F9 FF      |
| PICTURE       | 250 128        | FA 80      |
| WAVE          | 250 129        | FA 81      |
| LINETO        | 251 94         | FB 5E      |
| FILLPOLY      | 251 210        | FB D2      |
| INVERTPOLY    | 251 211        | FB D3      |
| ERASEPOLY     | 251 212        | FB D4      |
| PAINTPOLY     | 251 213        | FB D5      |
| FRAMEPOLY     | 251 214        | FB D6      |
| PTAB          | 251 215        | FB D7      |
| FILLARC       | 251 216        | FB D8      |
| INVERTARC     | 251 217        | FB D9      |
| ERASEARC      | 251 218        | FB DA      |
| PAINTARC      | 251 219        | FB DB      |
| FRAMEARC      | 251 220        | FB DC      |
| FILLROUNDRECT | 251 221        | FB DD      |

| <b>TOKEN</b>    | <b>DECIMAL</b> | <b>HEX</b> | <b>TOKEN</b> | <b>DECIMAL</b> | <b>HEX</b> |
|-----------------|----------------|------------|--------------|----------------|------------|
| INVERTROUNDRECT | 251 222        | FB DE      | TEXTFONT     | 251 239        | FB EF      |
| ERASEROUNDRECT  | 251 223        | FB DF      | MOVE         | 251 241        | FB F1      |
| PAINTROUNDRECT  | 251 224        | FB E0      | MOVETO       | 251 242        | FB F2      |
| FRAMEROUNDRECT  | 251 225        | FB E1      | PENNORMAL    | 251 243        | FB F3      |
| FILLOVAL        | 251 226        | FB E2      | PENPAT       | 251 244        | FB F4      |
| INVERTOVAL      | 251 227        | FB E3      | PENMODE      | 251 245        | FB F5      |
| ERASEOVAL       | 251 228        | FB E4      | PENSIZE      | 251 246        | FB F6      |
| PAINTOVAL       | 251 229        | FB E5      | GETPEN       | 251 247        | FB F7      |
| FRAMEOVAL       | 251 230        | FB E6      | SHOWPEN      | 251 248        | FB F8      |
| FILLRECT        | 251 231        | FB E7      | HIDEPEN      | 251 249        | FB F9      |
| INVERTRECT      | 251 232        | FB E8      | OBSURECURSOR | 251 250        | FB FA      |
| ERASERECT       | 251 233        | FB E9      | SHOWCURSOR   | 251 251        | FB FB      |
| PAINTRECT       | 251 234        | FB EA      | HIDECURSOR   | 251 252        | FB FC      |
| FRAMERECT       | 251 235        | FB EB      | SETCURSOR    | 251 253        | FB FD      |
| TEXTSIZE        | 251 236        | FB EC      | INITCURSOR   | 251 254        | FB FE      |
| TEXTMODE        | 251 237        | FB ED      | BACKPAT      | 251 255        | FB FF      |
| TEXTFACE        | 251 238        | FB EE      |              |                |            |





## A Fat Bits Approach to Cursor Editing

Let's take a look at one of the Macintosh ROM routines which can be called from MSBASIC (version 2.0). By using the method described here, you can customize your own cursors for use within your programs.

Pages 298 and 299 of the MS BASIC manual explain the Mouse Cursor Handling Routines. The program 'Cursor Editor' in this article demonstrates how to build your own cursor using the `CALL SETCURSOR (VARPTR( cursor % (0)))` function. It would be helpful to enter the program and run it as you read the explanation here.

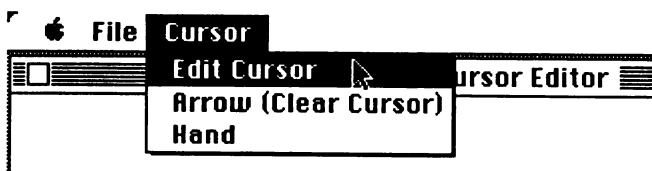


Fig. 1 Cursor Edit Program Menu

When you run the program, the main BASIC menus are replaced by a new File and Cursor menu. The Cursor menu allows you to edit the current custom cursor. The Arrow is the default cursor and cannot be edited unless you duplicate it as a custom cursor of your own. Choosing 'Hand' sets the cursor to a hand. An examination of the 'Hand' routine at the end of the program may help you to understand how the `cursor%(0)` array is set up.

By selecting 'Arrow' in the cursor menu you may clear the cursor to be edited to a blank. The editor is set to a blank (i. e. no cursor) when the program starts. Select 'Edit Cursor' from the menu. The bit parameters for the screen grid are set up and a blank grid is printed on the screen. (See Fig. 2,

lower left corner).

Now just point and click the mouse to select what the cursor will look like. The program will appear to run a bit slow here while the correct bit is being selected. This takes longer in Basic than in most other languages, so be patient. When the cursor is finished click the OK button.

Now a new grid comes up to create a mask for our new cursor. The cursor mask selects which bits behind the cursor (the entire 16X16 grid). The program allows you to experiment with different masks until you get it just right. To test it out you can move the cursor over a black area of the screen and then over a white area and see what part of the background is allowed to be seen through the cursor. It is purely subjective so you can keep on trying till you are happy with it. For our example, I have filled in the inside of the cursor to make the mask.

Any pixels of the screen which are behind the mask will not be allowed to be displayed. Click the OK button to continue. Next we set the hot spot of the cursor. The hot spot is the active area of the cursor that determines where it is pointing to. It is the intersection of the corners of the pixels. For our example, we want the hot spot to be at the tip of the pencil so we click there. A small square appears to mark the spot. You may want to try other locations for the hot spot if you don't quite understand the significance of the hot spot. Click OK to continue. Now the new cursor will appear. **WARNING:** If the cursor which was defined had no pixels selected (i. e. a blank cursor), then it will be very difficult to select anything from this point on, because you will not be able to see the cursor. If this happens you may have to abort (use command-period to halt a basic program). Basic will reinitialize the arrow cursor when you exit the program.

Your new cursor can now be stored in a file on the disk by using the file menu and loaded at a later time. The load routine may be copied for use in your own program to read in a custom cursor for your own programs. After the cursor data is stored to disk, the program will change the file type to "CURS" to enable the load routine to recognize only those files which contain cursor data. Any filename type may be changed so that the routine which reads the data may only select files of the same type.

Hopefully this program provides an easy way to customize your own cursors. A good way to create a library of cursors is to do a screen dump to disk of the MacPaint screen. Then load the dumped

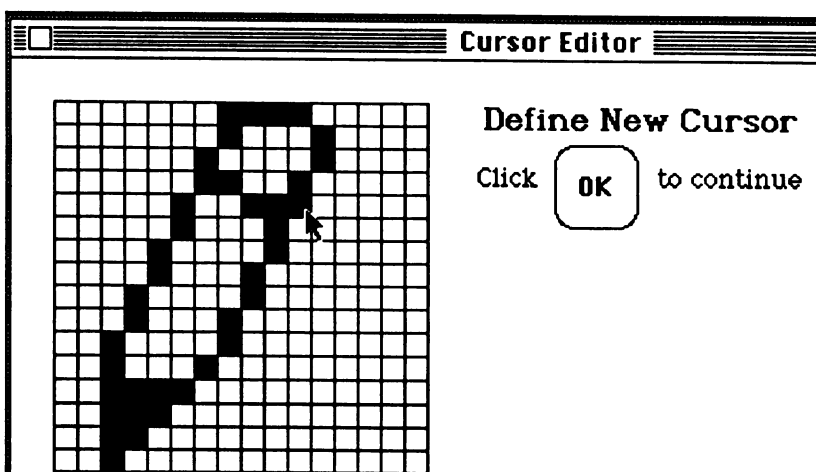


Fig 2 Creating a New Cursor

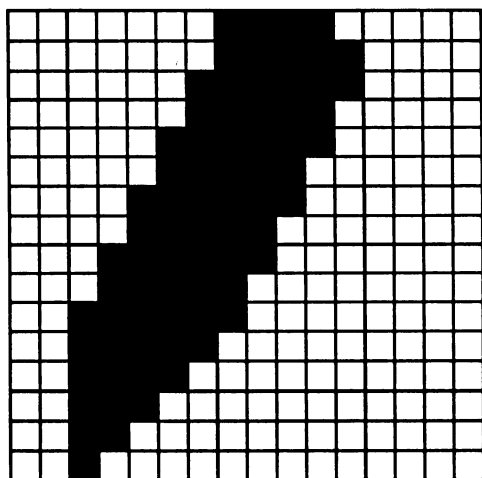


Fig. 3 Creating the Mask

screen and look at the cursors used by MacPaint by using FatBits and dump each of them to the printer from the Fat Bit screen. Then the cursor can be copied bit by bit into the cursor editor program and saved. The program is available on disk if you don't want to have to type. Have fun!!!

' Cursor Editor  
' By Dave Kelly  
' ©MACTUTOR 1985

```
DEFINT i,j,k
DIM Cursor%(34), Bstatus%(512),
 Bound0%(256), Bound1%(256),
 Bound2%(256), Bound3%(256)
editor%=0:NewYork=2:Bold=1
Plain=0:Geneva=3
WINDOW 1,"Cursor Editor",
 (2,40)-(510,340),1
' Erase BASIC menus
FOR i=3 TO 5:MENU i,0,0,"":NEXT i
MENU 1,0,1,"File"
MENU 1,1,1,"Load Cursor"
MENU 1,2,1,"Save Cursor"
MENU 1,3,1,"Quit"
MENU 2,0,1,"Cursor"
MENU 2,1,1,"Edit Cursor"
MENU 2,2,1,"Arrow (Clear Cursor)"
MENU 2,3,1,"Hand"
```

```
ON MENU GOSUB Checkmenu: MENU ON
IF editor%=0 THEN MENU STOP:GOSUB
 InitEditor:MENU ON
loop:GOTO loop
```

```
Checkmenu:
 menunumber=MENU(0)
```

## Define New Mask

Click **OK** to continue

```
menuitem=MENU(1):MENU
IF menunumber=1 THEN filemenu
ON menuitem GOSUB
 Editor,Arrow,Hand
RETURN
```

filemenu:

```
ON menuitem GOSUB load.cursor,
 save.cursor,quit
```

RETURN

InitEditor:

```
x=20:y=20:offsetx=0:offsety=0:editor%=1
```

```
TEXTFONT(NewYork)
```

```
TEXTSIZE(14):TEXTFACE(Bold)
```

```
LOCATE 5,1
```

```
PRINT"Please wait.... Initializing
 Editor."
```

```
MENU 1,0,0:MENU 2,0,0
```

```
FOR i=0 TO 33: cursor%(i)=0:NEXT i
```

```
PICTURE ON
```

```
FOR j= 0 TO 15
```

```
FOR k=15 TO 0 STEP -1
```

```
rectangle%(0)=y+offsety
```

```
bound0%((j*16)+k)=rectangle%(0)
```

```
rectangle%(1)=x+offsetx
```

```
bound1%((j*16)+k)=rectangle%(1)
```

```
rectangle%(2)=y+offsety+12
```

```
bound2%((j*16)+k)=rectangle%(2)
```

```
rectangle%(3)=x+offsetx+12
```

```
bound3%((j*16)+k)=rectangle%(3)
```

```
bstatus%((j*16)+k)=0
```

```
offsetx=11+offsetx
```

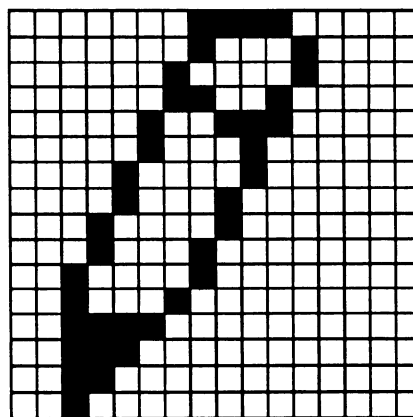


Fig. 4 Setting the Hot Spot



```

FRAMERECT(VARPTR(rectangle%(0)))
 NEXT k
 offsety=11+offsety:offsetx=0
NEXT j
PICTURE OFF
grid$=PICTURE$
MENU 1,0,1:MENU 2,0,1:CLS
RETURN

```

Editor:

```

MENU 1,0,0:MENU 2,0,0
TEXTFONT(NewYork)
TEXTSIZE(14):TEXTFACE(Bold)
LOCATE 5,1:PRINT"Please wait for
 Setup of Editor."
GOSUB Bitstatus

```

' set up new cursor

```

GOSUB print.pic
LOCATE 2,26:PRINT"Define New
 Cursor"

```

```

GOSUB Print.message
GOSUB Draw.Datapixels
GOSUB Define

```

' set up new mask

```

GOSUB print.pic
LOCATE 2,26:PRINT"Define New Mask"
GOSUB Print.message
GOSUB Draw.Maskpixels
GOSUB Define

```

' set hot spot

```

GOSUB print.pic
LOCATE 2,26:PRINT"Set Hot Spot"
GOSUB Print.message
GOSUB define.hotspot
CLS:BUTTON CLOSE 1
TEXTFONT(NewYork):TEXTSIZE(14)
TEXTFACE(Bold)
LOCATE 5,1:PRINT"Please wait."
GOSUB Cursor.done
SETCURSOR (VARPTR(cursor%(0)))
CLS:MENU 1,0,1:MENU 2,0,1
RETURN

```

Print.pic:

```

CLS:PICTURE,grid$
TEXTFONT(NewYork)
TEXTSIZE(14)
TEXTFACE(Bold)
RETURN

```

Print.message:

```

TEXTFACE(Plain)
TEXTSIZE(12)
LOCATE 4,35:PRINT"Click to continue" 'Note:
Space must be ^^^^^ here
BUTTON 1,1,"OK",
 (310,40)-(350,80),1
RETURN

```

define.hotspot:

```

GOSUB Draw.Datapixels
CALL PENSIZE(4,4):CALL PENMODE(10)
CALL MOVETO((cursor%(33)*11)+x,
 (cursor%(32)*11)+y)
CALL LINE(0,0)
WHILE DIALOG(0)<>1
IF MOUSE(0)>0 THEN GOSUB hotspot
WEND
CALL PENNORMAL
BEEP
RETURN

```

hotspot:

```

xpos=MOUSE(1):ypos=MOUSE(2)
IF xpos<x THEN xpos=x
IF ypos<y THEN ypos=y
IF xpos>x+16*11 THEN xpos=x+16*11
IF ypos>y+16*11 THEN ypos=y+16*11

```

```

CALL LINE(0,0)
cursor%(33)=INT((xpos-x)/11)
cursor%(32)=INT((ypos-y)/11)
CALL MOVETO((cursor%(33)*11)+x,
 (cursor%(32)*11)+y)
CALL LINE(0,0)
RETURN

```

Draw.Maskpixels:

```

maskpixel=1
FOR i= 256 TO 511
 IF bstatus%(i)=1 THEN
 rectangle%(0)=bound0%(i-256):
 rectangle%(1)=bound1%(i-256):
 rectangle%(2)=bound2%(i-256):
 rectangle%(3)=bound3%(i-256):
 PAINTRECT(VARPTR
 (rectangle%(0))):
 FRAMERECT(VARPTR
 (rectangle%(0)))
NEXT i
RETURN

```

Draw.Datapixels:

```

maskpixel=0
FOR i= 0 TO 255
 IF bstatus%(i)=1 THEN
 rectangle%(0)=bound0%(i):
 rectangle%(1)=bound1%(i):
 rectangle%(2)=bound2%(i):
 rectangle%(3)=bound3%(i):
 PAINTRECT(VARPTR
 (rectangle%(0))):
 FRAMERECT(VARPTR
 (rectangle%(0)))
NEXT i
RETURN

```

mousepress:

```

GOSUB getpixel 'see which pixel is selected
IF pixel%=256 THEN RETURN
IF Bstatus%(Pixel%+maskpixel*256)=0

```

```

 THEN Bstatus%(Pixel%+
 maskpixel*256)=1 ELSE
 Bstatus%(Pixel%+
 maskpixel*256)=0
rectangle%(0)=bound0%(Pixel%)
rectangle%(1)=bound1%(Pixel%)
rectangle%(2)=bound2%(Pixel%)
rectangle%(3)=bound3%(Pixel%)
INVERTRECT(VARPTR(rectangle%(0)))
FRAMERECT(VARPTR(rectangle%(0)))
RETURN

getpixel:
Pixel%=256
FOR i = 0 TO 255
 IF bound0%(i)<MOUSE(2) AND
 bound2%(i)>MOUSE(2) AND
 bound1%(i)<MOUSE(1) AND
 bound3%(i)>MOUSE(1) THEN
 Pixel%=i:256

NEXT i
RETURN

Cursor.done:
FOR j=0 TO 31
 cursor%(j)=0
 FOR k=14 TO 0 STEP -1
 cursor%(j)=(Bstatus%((j*16)+k)*(2^k))+
 cursor%(j)
 NEXT k
 IF Bstatus%((j*16)+15)=1 THEN
 cursor%(j)=cursor%(j)+&H8000
NEXT j
RETURN

Define:
WHILE DIALOG(0)<>1
 IF MOUSE(0)>0 THEN GOSUB
 mousepress
WEND
BEEP
RETURN

Bitstatus:
FOR j=0 TO 31
 t%=cursor%(j)
 FOR k=15 TO 0 STEP -1
 IF t%<0 THEN
 Bstatus%((j*16)+k)=1:t%=
 t%-&H8000:GOTO endloop
 IF t%<2^k THEN
 Bstatus%((j*16)+k)=0
 IF t%>=2^k THEN
 Bstatus%((j*16)+k)=1:t%=
 t%-2^k
 endloop:NEXT k
 NEXT j
RETURN

load.cursor:
filename$=FILES$(1,"CURS")
IF filename$="" THEN exitload

```

```

OPEN filename$ FOR INPUT AS #1
FOR i= 0 TO 33
 INPUT #1,cursor%(i)
NEXT i
CLOSE #1
exitload:CALL SETCURSOR(VARPTR(cursor%(0)))
RETURN

save.cursor:
filename$=FILES$(0)
IF filename$="" THEN exitsave
OPEN filename$ FOR OUTPUT AS #1
FOR i=0 TO 33
 PRINT #1,cursor%(i)
NEXT i
CLOSE #1
NAME filename$ AS filename$,"CURS"
exitsave:CALL SETCURSOR (VARPTR
 (cursor%(0)))
RETURN

quit:
 BUTTON CLOSE 1:MENU RESET
TEXTFONT(Geneva):TEXTFACE(Plain)
TEXTSIZE(12)
END

Arrow:
 INITCURSOR
 FOR i=0 TO 33: cursor%(i)=0:NEXT i
 RETURN

Hand:
 ' Cursor Data
 Cursor%(0)=&H0:Cursor%(1)=&H0
 Cursor%(2)=&H700:Cursor%(3)=&H1900
 Cursor%(4)=&H2200:Cursor%(5)=&H4700
 Cursor%(6)=&HC7FE:Cursor%(7)=&H8C01
 Cursor%(8)=&H97FE:Cursor%(9)=&HE410
 Cursor%(10)=&H87E0:Cursor%(11)=&H8420
 Cursor%(12)=&HC7C0:Cursor%(13)=&H7F80
 Cursor%(14)=&H0:Cursor%(15)=&H0
 ' Cursor Mask
 Cursor%(16)=&H0:Cursor%(17)=&H0
 Cursor%(18)=&H700:Cursor%(19)=&H1F00
 Cursor%(20)=&H3E00:Cursor%(21)=&H7F00
 Cursor%(22)=&HFFFE:Cursor%(23)=&HFFFF
 Cursor%(24)=&HFFFE:Cursor%(25)=&HFFF0
 Cursor%(26)=&HFFE0:Cursor%(27)=&HFFE0
 Cursor%(28)=&HFFC0:Cursor%(29)=&H7F80
 Cursor%(30)=&H0:Cursor%(31)=&H0
 Cursor%(32)=7 'Vertical hot spot
 Cursor%(33)=16 'Horizontal hot spot
 SETCURSOR(VARPTR(cursor%(0)))
 RETURN

```



*For the Finest in Technical  
information, subscribe to  
MacTutor, The Macintosh  
Programming Journal.*

# Basic School

## Reading Paint Files



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 10

This month we will explore how to read MacPaint files via MSBASIC. We will start by generally dissecting a paint file.

The 8"x10" MacPaint pictures that we are used to seeing are represented by 576x720 pixels. That's 414,720 pixels, which would require 51,840 bytes to store directly to disk. The first 512 bytes (first block) contain the brush and fill pattern information. Fortunately, the bit map is compressed a row at a time in order to conserve disk space, or we really wouldn't have much room to do anything else. Therefore there is a block of encoded pixels for each of the 720 rows. There are two flavors of encoding these entries, a straight bitmap and a run-length encoded sequence.

### Pattern Bytes

The first 512 bytes start with 00 00 00 02. After these four bytes come the 38 patterns that you see when you open your MacPaint document. The patterns may be edited with the pattern editor provided in MacPaint and your custom patterns may then be used as needed. Each of the 38 patterns are represented by 8 bytes each, where each byte represents a row in the pattern grid. The pattern rows are mapped from left to right and top to bottom. The order of the patterns is the same as appear at the bottom of the MacPaint screen. The patterns are not encoded. The remainder of the first 512 bytes are filled with zeros.

### Bitmap Flavors

Both flavors start with a one byte count. In the bitmap flavor the count indicates the number of pixel bytes which will follow minus one. As an example of this see figure 1 (Flavor 1) below:

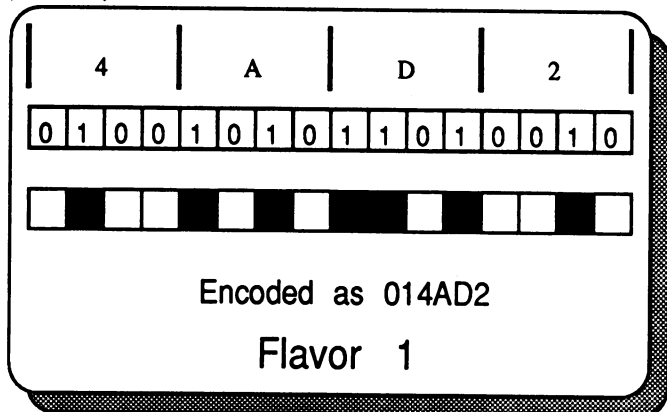


Figure 1: Bit Map Flavor

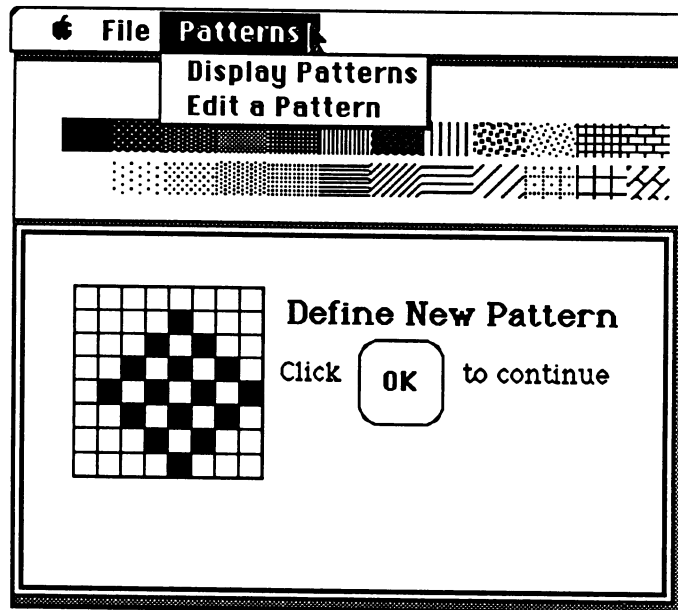


Fig. 2 Program Output

In the example, the pattern 0100101011010010 (where a 1 represents a black pixel and a 0 represents a white pixel) would be encoded as 014AD2. The first byte (01) is the count minus one, therefore the count is 2 bytes. The following 2 bytes represent the data (4AD2).

The second flavor is composed of a sequence of 8 bits which is repeated. The count indicates the number of times which the sequence is repeated. For the second flavor the count byte (first byte) is set negative. Negative binary numbers have the first bit set to indicate that it is negative. In this case, the first byte is the absolute value of the of the first byte minus one. For example FDFD would represent 4 bytes of "FF", or "FF FF FF FF". FD is 11111101. The absolute value (negative) is 00000010 = 3. Therefore the count is 3+1 = 4 bytes. A row that is entirely blank is represented by B900 where B9 = -71. The inverse is 71. Add one to get the count (71+1=72 bytes of the 8-bit pattern "00". There are 72 bytes (72X8 = 576 pixels) in one row of a MacPaint document. The pattern may be any combination of ones and/or zeros. This would be most useful in coding patterns of bits which are repeated.

The tricky part is determining which of the flavors to use. In reading the file this is not too much of a problem, because we can always look at the first byte. If it is negative then it is of the second flavor. Writing a MacPaint file is not quite as simple but can be done if you plan ahead.

## Basic Programs

I'm sure that you have probably seen the public domain program which will read the top left corner of a MacPaint document and display it on the screen. The program below called Paint Pokes will read any MacPaint file and display it on the screen using the Prof. Mac Pokes procedures that I have shown before (see Apr. 85 MacTutor, pg. 34). The problem is that this method of printing on the screen is extremely slow. I would still like to convert the paint file (or a portion of it) into a BASIC PICTURE\$ which could be printed to the screen quicker, but so far I have not been able to understand how PICTURE\$ (or the clipboard for that matter) is encoded. The problem is that BASIC does not recognize anything that has been poked onto the screen. Because of this, the screen GET and screen PUT won't work for saving the picture via BASIC for later use. Anyone have the answer??

Paint Pokes asks for the paint file to be displayed and then reads the file and pokes it onto the screen. Press the mouse button at the end to continue when the display has finished (or when you have finished looking at it). The program then returns to BASIC.

Pattern Editor is a BASIC program which allows you to edit the patterns used in MacPaint. The pattern editor built into MacPaint is really more useful, but the BASIC pattern editor demonstrates the format of the patterns stored in a paint file. To use the pattern editor, run the program and select the desired menus. First load in a Paint document. The entire document is read in so that it can be completely written back to disk again with a modified set of patterns. Select 'Display Pattern' to show the patterns as they are now defined. Choose 'Edit Pattern' to edit a particular pattern. Use the mouse to select the desired pattern and then a grid will appear and allow you to change the bits. To change the pattern as you are editing it choose 'Display Pattern' and the patterns will be re-drawn using your new pattern. If you display the pattern, the pattern will now be changed for

good, you can't undo what has been displayed unless you want to re-edit the same pattern back to the original pattern. You may revert back to the original pattern set by loading the paint document again or you may want to save the changes to a new Paint document.

Thanks go to Bob Denny for his analysis of MacPaint file format.

' **Paint Pokes**  
' By Dave Kelly  
' ©MACTUTOR 1985

```
start=108288! 'Set up beginning
screen addr.
ending=130175! 'Set up ending
screen addr.
mac512=(512-128)*1024 'Set up addr.
for 512K Mac
IF FRE(0)>100000! THEN
start=start+mac512
ending=ending+mac512
screen=start
WINDOW 1,"",(0,0)-(512,342),3
x$=FILES$(1,"PNTG") 'Get a
MacPaint file
IF x$="" THEN quit 'No selection, quit
HIDECURSOR
OPEN x$ FOR INPUT AS #1
FOR i%= 1 TO 512 'Discard the first
512 bytes
x$=INPUT$(1,#1)
NEXT i%
pixel%=0
WHILE NOT EOF(1)
count=ASC(INPUT$(1, #1))
IF count<&H80 THEN GOSUB
type1 ELSE GOSUB type2 'Check if
high bit is set
WEND
CLOSE #1
```

```
wait.for.mouse.click:
pause:IF MOUSE(0)>0 THEN quit
GOTO pause
```

```
Quit: MENU
RESET:SHOWCURSOR
WINDOW 1,"Output Window",(2,40)-
(510,340),1
END
```

```
Pokescreen:
IF pixel%>511 THEN RETURN
POKE screen,ASC(byte$)
screen=screen+1
IF screen >=ending THEN
wait.for.mouse.click
```

## RETURN

```
type1: 'first flavor
FOR i% = 1 TO count+1
byte$=INPUT$(1,#1) 'Read a byte
GOSUB Pokescreen
pixel%=pixel%+8 'Count pixels
printed
NEXT i%
IF pixel%>=576 THEN pixel%=0
'line is full
RETURN
```

```
type2: 'second flavor
byte$=INPUT$(1,#1) 'Read a byte
FOR i%=1 TO (&H101-count)
GOSUB Pokescreen
pixel%=pixel%+8 'Count pixels
printed
NEXT i%
IF pixel%>=576 THEN pixel%=0 'line
is full
RETURN
```

```

' Pattern Editor
' By Dave Kelly
' ©MACTUTOR 1985
```

```
start=108288! 'Set up beginning screen
addr.
ending=130175! 'Set up ending
screen addr.
mac512=(512-128)*1024 'Set up
address for 512K Mac
IF FRE(0)>100000! THEN
start=start+mac512:ending=ending+ma
c512
screen=start:editor%=0:NewYork=2
Bold=1:Plain=0: DEFINT i,j,k
DIM pattern%(512),k$(1),
Bound0%(72),Bound1%(72),
Bound2%(72),Bound3%(72),
Bstatus%(72),
```

```
WINDOW 1,"",(2,24)-(510,150),3
FOR i%=1 TO 5
MENU i%,0,0,"" 'Erase old menus
NEXT i%
MENU 1,0,1,"File"
MENU 1,1,1,"Load Paint patterns"
MENU 1,2,0,"Save Paint patterns"
MENU 1,3,1,"Quit"
MENU 2,0,0,"Patterns"
MENU 2,1,1,"Display Patterns"
MENU 2,2,1,"Edit a Pattern"
```

```
ON MENU GOSUB menu.selection
MENU ON
pause:GOTO pause
```

```

menu.selection:
 menunumber=MENU(0)
 menuitem=MENU(1):MENU:MENUE OFF
 IF menunumber=1 THEN ON menuitem
 GOSUB read.paint.file,
 write.paint.file, Quit
 IF menunumber=2 THEN ON menuitem
 GOSUB Display.patterns,
 Edit.pattern
 MENU ON:RETURN

read.paint.file:
 x$=FILES$(1,"PNTG") 'Get a MacPaint file
 IF x$="" THEN RETURN 'No selection,
 then forget it.

 MENU 1,0,0:MENUE 2,0,0
 HIDECURSOR
 TEXTFONT(NewYork):TEXTSIZE(14)
 TEXTFACE(Bold)
 CLS:LOCATE 4,1
 PRINT"Now Reading Macpaint document.....
 Please wait."
 OPEN x$ FOR INPUT AS #1 LEN= 2000
 ERASE k$
 DIM k$(LOF(1)+1)
 FOR i%= 1 TO 512
 k$(i%)=INPUT$(1,#1)
 NEXT i%
 WHILE NOT EOF(1)
 k$(i%)=INPUT$(1,#1)
 i%=i%+1
 WEND
 number.of.bytes=i%
 CLOSE #1
 SHOWCURSOR:CLS
 MENU 2,0,1:MENUE 1,0,1:MENUE 1,2,1
 BEEP:RETURN

write.paint.file:
 TEXTFONT(NewYork):TEXTSIZE(14)
 TEXTFACE(Bold)
 CLS:LOCATE 4,1
 PRINT "Now writing new MacPaint document....
 Please wait."
 x$=FILES$(0,"Choose a new filename:") 'Get a
 Paint file
 IF x$="" THEN RETURN 'No selection, then
 forget it.

 MENU 2,0,0:MENUE 1,0,0
 OPEN x$ FOR OUTPUT AS #1
 FOR i1%=1 TO number.of.bytes
 PRINT #1, k$(i1%);
 NEXT i1%
 CLOSE #1
 NAME x$ AS x$,"PNTG"
 MENU 1,0,1:MENUE 2,0,1
 LOCATE 4,1:PRINT SPACES$(80)
 BEEP:RETURN

Quit: MENU RESET:SHOWCURSOR
WINDOW CLOSE 2
WINDOW 1,"Output Window",

```

```

 (2,40)-(510,340),1
END

Display.patterns:
'Poke the display onto the screen
HIDECURSOR:MENUE 1,0,0:MENUE 2,0,0
a1=start+64*50+3
b1=3
'These loops poke the patterns to the screen
FOR qloop =0 TO 9 STEP 8
FOR pat% = 0 TO 18
 FOR i%= (pat%*8)+1 TO (8*pat%)+8
 row=i%+4
 POKE a1+((row MOD 8+qloop)*64)+
 (b1*pat%), ASC(k$(row))
 POKE a1+((row MOD 8+qloop)*64)+
 (b1*pat%)+1, ASC(k$(row))
 POKE a1+((row MOD 8+qloop)*64)+
 (b1*pat%)+2, ASC(k$(row))
 NEXT i%
NEXT pat%
NEXT qloop
a1=start+64*70+3
FOR qloop=0 TO 9 STEP 8
FOR pat% = 19 TO 38
 FOR i%= (pat%*8)+1 TO (8*pat%)+8
 row=i%+4
 POKE a1+((row MOD 8+qloop)*64)+
 (b1*(pat%-19)), ASC(k$(row))
 POKE a1+((row MOD 8+qloop)*64)+
 (b1*(pat%-19))+1, ASC(k$(row))
 POKE a1+((row MOD 8+qloop)*64)+
 (b1*(pat%-19))+2, ASC(k$(row))
 NEXT i%
NEXT pat%
NEXT qloop
SHOWCURSOR:MENUE 1,0,1:MENUE 2,0,1
RETURN

Choose.pattern: 'Pattern% = selected pattern #
SHOWCURSOR:WINDOW 1
IF editor%=1 THEN makeselection
GOSUB Display.patterns
makeselection:
LOCATE 5,1:PRINT"Please use mouse to
 select pattern"

Edit.pattern:
GOSUB Choose.pattern
IF editor%=1 THEN editor
InitEditor:
WINDOW 2,"Pattern Editor",
 (10,160)-(500,320),2
x=20:y=20:offsetx=0:offsety=0:editor%=1
TEXTFONT(NewYork):TEXTSIZE(14)
TEXTFACE(Bold)
LOCATE 5,1
PRINT"Please wait.... Initializing Editor."
PICTURE ON
FOR j= 0 TO 7
 FOR k=7 TO 0 STEP -1
 rectangle%(0)=y+offsety

```

```

Bound0%((j*8)+k)=rectangle%(0)
rectangle%(1)=x+offsetx
Bound1%((j*8)+k)=rectangle%(1)
rectangle%(2)=y+offsety+12
Bound2%((j*8)+k)=rectangle%(2)
rectangle%(3)=x+offsetx+12
Bound3%((j*8)+k)=rectangle%(3)
Bstatus%((j*8)+k)=0:offsetx=11+offsetx
FRAMERECT(VARPTR(rectangle%(0)))
NEXT k
offsety=11+offsety:offsetx=0
NEXT j
PICTURE OFF
grid$=PICTURE$

editor:
WINDOW 2
TEXTFONT(NewYork):TEXTSIZE(14)
TEXTFACE(Bold)
LOCATE 5,1
PRINT"Please wait for Setup of Editor."
GOSUB Bitstatus
'set up new cursor
GOSUB print.pic
LOCATE 2,26:PRINT"Define New Pattern"
GOSUB Print.message
GOSUB Draw.Datapixels
GOSUB Define
CLS:BUTTON CLOSE 1
TEXTFONT(NewYork):TEXTSIZE(14)
TEXTFACE(Bold)
LOCATE 5,1:PRINT"Please wait."
GOSUB Set.pattern
LOCATE 5,1:PRINT SPACE$(70)
GOSUB Display.patterns
RETURN

print.pic: CLS:PICTURE,grid$ 'print the grid
TEXTFONT(NewYork)
TEXTSIZE(14)
TEXTFACE(Bold)
RETURN

Print.message:
TEXTFACE(Plain)
TEXTSIZE(12)
LOCATE 4,35
PRINT"Click to continue"
 "This space ^^^^belongs here"
BUTTON 1,1,"OK",(310,40)-(350,80),1
RETURN

Draw.Datapixels:
FOR i= 0 TO 63
 IF Bstatus%(i)=1 THEN rectangle%(0)=Bound0%(i)
:
 rectangle%(1)=Bound1%(i):
 rectangle%(2)=Bound2%(i):
 rectangle%(3)=Bound3%(i):
 PAINTRECT(VARPTR(rectangle%(0))):
 FRAMERECT(VARPTR(rectangle%(0)))

```

```

'end if
NEXT i
RETURN

mousepress:
GOSUB getpixel 'see which pixel is selected
IF pixel%=64 THEN RETURN
IF Bstatus%(pixel%)=0 THEN
 Bstatus%(pixel%)=1
ELSE Bstatus%(pixel%)=0
 rectangle%(0)=Bound0%(pixel%)
 rectangle%(1)=Bound1%(pixel%)
 rectangle%(2)=Bound2%(pixel%)
 rectangle%(3)=Bound3%(pixel%)
 INVERTRECT(VARPTR(rectangle%(0))):
 FRAMERECT(VARPTR(rectangle%(0))):
RETURN

getpixel:
 pixel%=64
 FOR i = 0 TO 63
 IF Bound0%(i)<MOUSE(2) AND
 Bound2%(i)>MOUSE(2) AND
 Bound1%(i)<MOUSE(1) AND
 Bound3%(i)>MOUSE(1) THEN
 pixel%=i:i=64

 NEXT i
RETURN

Define:
WHILE DIALOG(0)<>1
 IF MOUSE(0)>0 THEN GOSUB
 mousepress
 IF MENU(0)=2 AND MENU(1)=1 THEN
 GOSUB Set.pattern:
 GOSUB Display.patterns
WEND
BEEP:RETURN

Bitstatus: 'get status of each grid row
FOR j= 0 TO 7
 row=j+((pattern%-1)*8)+5
 t%=ASC(k$(row))
 FOR k=7 TO 0 STEP -1
 IF t%<2^k THEN Bstatus%(j*8+k)=0
 IF t%>=2^k THEN
 Bstatus%(j*8+k)=1:t%=t%-2^k
 endloop:NEXT k
 NEXT j
RETURN

Set.pattern: 'set bits to match bit status
FOR j= 0 TO 7
 row=j+((pattern%-1)*8)+5
 t%=0
 FOR k=7 TO 0 STEP -1
 t%=(Bstatus%(j*8+k)*2^k)+t%
 NEXT k
 k$(row)=CHR$(t%)
NEXT j
RETURN

```



# Special Projects

## Rescue that Protected Basic Program!

Mike Steiner  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 10

In my diggings into Microsoft BASIC, I found some interesting things about how a program is stored on disk. My ultimate aim is to be able to "crack" a protected BASIC program; I once lost a program because I had saved it as protected and did not have an unprotected back-up copy.

The information in this column is valid only for BASIC Version 2.0; the discussion and program are not valid for Version 2.1. Most of the data presented in 'MS BASIC Tokens Explained' in the July 1985 MacTutor do not apply to Version 2.1. Also, the program is for a 512K Macintosh. Change the starting memory location (Peekloc) for a 128K or 1024K Macintosh as described in 'MS BASIC Tokens Explained.'

Each version of BASIC (Binary and Decimal) is coded so that the finder treats them as two different creators. This, I assume, is because of the incompatibilities between the two versions. See last month's discussion of how numbers are stored in the two versions.

Table 1 shows the creators, file types, and ID bytes of the various files used by Microsoft BASIC version 2.0. (See sidebar for a brief explanation of creators and file-types.) The four-letter codes in parentheses on the top line of the table are the identification of the creators of the files. The four-letter codes in the table show the file types. The hexadecimal numbers in parentheses show the first byte (ID byte) of the files. TEXT files do not have an ID byte.

Once a file is loaded, the ID byte tells BASIC whether the file is protected. TEXT files do not need an ID byte because BASIC knows it is a text file and loads it in as is, translating reserved words into their hexadecimal tokens. BASIC can load any text file and treat it as a program listing (sometimes with curious results if the

file is not a program). Each version (binary or decimal) can load compressed and protected files that were created only by itself. However, you can use SetFile to change the creator ID to load a file from the 'other' BASIC. If you do this, I recommend changing the ID byte with a disk zap program such as fEdit. Keep in mind that if there are any non-integer numbers defined in the program listing, their format will not be proper for the 'new' creator.

Protected programs, in addition to being having a unique TYPE and ID byte, are encyphered. It appears that this encyphering is only in the Macintosh version of Microsoft BASIC; friends with CP/M and MS DOS versions have not encrypted files with their versions of Microsoft BASIC. I have not yet been able to decypher these files; I have however, devised an interesting solution!

Following is a discussion of how you can prevent losing access to protected programs. The method discussed will not work on programs already protected; it will, however, give you a 'back door' into future programs. It works because when a protected program is loaded, it is decyphered and resides in memory as if it had been loaded from a text or compressed file. However, LIST, PEEK, and a slew of other commands are disabled in the command window. These commands have not been disabled from running from within the program, and that is what allows us to recover the file.

Listing 1 is a subprogram called "rescueit" which may be appended to any program prior to saving it as protected. Save rescueit as a TEXT file and MERGE it to your program. To use it, type the name of the subprogram in the command window, answer one question, and wait until it says that it is finished. (The name of the subprogram is in effect a password, so I strongly

recommend that you change it to a name that only you know; otherwise anyone who knows it can recover your program.) Your program will be saved as a file called "Ressurrected."

Rescueit asks whether you are using Binary or Decimal BASIC. It then opens a file called "res" for output and then PRINTs one byte (the ID byte for a compressed program in the version of BASIC you are using) to the file. It then PEEKs the program in memory and PRINTs it to res, one byte at a time. It then PRINTs five zeroes at the end of the file. Read 'MS BASIC Tokens Explained' to see why this is necessary. After closing res, rescueit renames "res" as "Ressurrected" and changes the filetype from TEXT to MSBC or MSBB, as appropriate. Then it prints "finished.." on the screen and exits to the desktop. Calling a subprogram from the command window sometimes scrambles pointers and causes memory problems. By exiting to the finder, these problems are avoided. The next step is to re-enter BASIC, load "Ressurrected" using the Standard File and save the program. The program is now saved in compressed mode, ready to run or be edited. This last step sets the creator. If this step is omitted, the program will still load and run, but only from within BASIC; the finder will not recognize an application to run the program if it is double-clicked, and the file will have the generic document icon. The only difference between resurrected and the original program is that resurrected has an extra blank line at the end of the listing. If you understand how BASIC stores programs in memory, you should be able to figure out why this is so. The reason is printed in a question / answer session at the end of this article.

Use of the program is fairly straightforward. Assuming that you already have rescueit on your disk

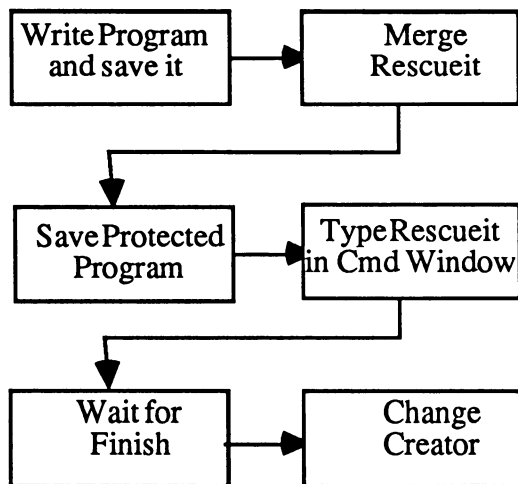


**Fig. 1 How Rescuelt Works**

(Remember to save it as a text file, not as a compressed program.), write your program and save it as a text or compressed file. (Don't forget to always keep a backup file!) In the command window, type "Merge "rescuelit"." This will cause rescuelit to be appended to the end of your program. Then choose "Save As..." from the File menu. Save your program (with rescuelit appended) as a protected program. Now, if you ever lose your backup and need to recover the protected file, load BASIC and open the protected file. Within the command window, type rescuelit (or the name to which you renamed the sub-program). The routine will execute, and when it ends will tell you that it is finished and return to the desktop. Next, set the creator as described above. Whenever you use rescuelit, start with the Macintosh turned off. As discussed in 'MS BASIC Tokens Explained,' if another application has been run before BASIC, sometimes BASIC programs do not load at the same memory location. Rescuelt always assumes that the program loads at the same point. Illustration 1 is a brief schematic of the rescuelit process.

### File Types and Creator Bytes

File-type and creator (also called file-type and creator flags) are four-letter abbreviations that show the file type and which application generated the file. An application (i.e. a program that can be run by double-clicking it's icon on the desktop) is a file that always has the file-type APPL. All other files are known as 'Documents' and may have any file-type that the programmer chooses.

Applications without a defined icon use the generic application icon (the diamond-shaped paper with a hand writing on it). Documents without a defined icon use the generic document icon (a piece of blank paper with the upper right corner folded down).

The finder uses the file type and creator flags to know which document files can be loaded by a given application. A document is linked to it's creator by the creator flag. When a document and a program have the same creator, the finder knows to launch that program when you double click on the document icon; once the program is launched, the finder tells

it to open the document. The file type flag has a number of purposes. The most common is to assign an icon to the file. All the desk-top icons used by a program are stored in it's ICN# resource. The finder uses the type and creator flags to assign the correct icon to the file. (However, for document files, a bit in the application called the bundle bit must be set to 1; otherwise, the link is not established.) Also, the Standard File (which contains the list of documents that you can open when you choose Open from the File menu within a program) uses the file type to determine which files may be opened. These files are not restricted to those created by the application; for example, Microsoft Word can open files created by Apple's Macwrite. However, you cannot open a Macwrite document with Word by double-clicking the document icon; you can only do it from within Word via the Standard File.

Two interesting features of Microsoft BASIC are that you can specify file types when loading a file from within a program by use of the FILE\$(1) function; by using the NAME command, you can change the file-type of a file on the disk. Rescuelt makes use of this latter feature to change the rescued program from a TEXT file to a BASIC compressed file; this step is necessary because the file is generated in the compressed format, but saved as a TEXT file.

So far as I have been able to determine, there is no way for a Microsoft BASIC program to determine which version of BASIC it is using. The prefix byte exists only on the disk, and is not stored in memory; FILE\$(1) can read the file type, but not the creator; NAME likewise can change only the file type and name. Does anyone know how to read or set the creator of a file from within MS BASIC? MacTutor will pay \$50 to the first submission that shows how to do it.

**Table 1**

Microsoft BASIC 2.0 Creator and File Type Codes

| Mode       | Binary (MSBB) | Decimal (MSBA) |
|------------|---------------|----------------|
| Text       | TEXT          | TEXT           |
| Compressed | MSBC (F9)     | MSBB (FB)      |
| Protected  | MSBD (F8)     | MSBP (FA)      |

### Listing 1 - Rescuelt

```

REM }}} Marker for end of main program - don't change!!
SUB rescuelit STATIC: REM Change the name "rescuelt" of
sub-program for password protection

```

```

prefixbyte(1) = &HF9:prefixbyte(2) = &HFB
filetype$(1) = "MSBC":filetype$(2) = "MSBB"
Peekloc(1) = 66999!: Peekloc(2) = 77001!

```

```

WINDOW 1,.(100,100)-(400,200),2
TEXTFONT 0
PRINT " Which version of BASIC are you using?"
TEXTFONT 1
BUTTON 1,1,"Binary",.(35,40)-(95,65)
BUTTON 2,1,"Decimal",.(170,40)-(240,65)

```



```

WHILE DIALOG (0)<>1:WEND
buttonpressed = DIALOG (1)
Peekloc = Peekloc(buttonpressed)

WINDOW 1,"rescuit",(2,39)-(508,338),1
OPEN "res" FOR OUTPUT AS #1
PRINT#1,CHR$(prefixbyte(buttonpressed));

rescue1:
peekloc = peekloc +1
IF PEEK (peekloc) = &HAF THEN IF PEEK
 (peekloc+1) = &H20 AND PEEK (peekloc+2)
 = ASC ("}") AND PEEK (peekloc+3) = ASC
 ("[" AND PEEK (peekloc+4) = ASC ("{"
 THEN rescue2: REM Find marker
PRINT #1, CHR$ (PEEK (peekloc));
GOTO rescue1

rescue2:
FOR j = 1 TO 4
 PRINT #1,CHR$(&H0);
NEXT j
CLOSE #1
NAME "res" AS "Resurrected", filetype$(buttonpressed)

CALL TEXTSIZE (127)
PRINT: PRINT "Finished..";
SYSTEM
END SUB

```

**Q:** "Why does rescuit add a line feed to the end of a file?"

**A:** When rescuit (or any other file) is merged to a program, it is appended to the end of the program, starting on a new line. This generates a carriage return after the last line of the program. Then when rescuit runs, it looks for the REM line that starts the rescuit routine, and saves the program up to the last byte before the REM. This last byte is the carriage return that was inserted when rescuit was merged. A carriage return at the end of a program is treated by BASIC as a blank line.



# Basic School

## Printing Techniques for Basic



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 11

### Managing BASIC Font Printing

Sometimes when talking about the Macintosh, people have asked "How many columns does the Macintosh have?" My answer to them is that it depends. Many of you like myself started out on other computers such as the Apple II.

Many of the early computers had only 40 columns of text display. This severely limited the usefulness of word processing software until the introduction of 80-col display plug-in boards. Back in those days it was very important to ask how many columns there were because of the limitations that are imposed by the lack of 80 column display. The Macintosh is an entirely different animal, though. Because of the ability of the Macintosh to display proportional spaced fonts on the screen just as it will look when printed, it is not applicable to refer to the Macintosh screen as an 80-col display. It is important in development of applications for the Macintosh that we realize the difference between the Macintosh and other computers when it comes to displaying and printing data. Some may have experienced some frustration in trying to format text when using proportional fonts. I'll try to discuss a few items to keep in mind when using proportional spacing. There are some advantages and disadvantages to be aware of. This month we will explore some of the ins and outs of printing in different size fonts.

First, let's take a look at the difference between proportional and monospaced fonts. The unit of measurement for the spacing is the size of the character "O" in the current font. For monospaced fonts the character "O" happens to be the same width as the rest of the characters in the font. For

the proportional spaced font, the "O" is an average width, but not necessarily the same width as the other characters. Notice that the spacing of all the uppercase characters for a monospaced font is the same as for the lower case characters. Also note that the proportional spaced "A" is wider than the "a".

The advantage of using the monospaced type of font is that you can always be sure of just where each character will be printed or displayed. In the proportional font, each of the letters are a different width depending on how much spacing each character requires. Now that we have a feel for proportional characters, try the following lines of BASIC and compare the results of printing with monospacing and proportional spacing:

```
' Sample program #1
' Dave Kelly
' MACTUTOR ©1985
```

```
l1$="Name #1":r1$="Name #2"
l2$="123 Main St.":r2$="456 Central Ave."
l3$="Anyplace, USA":R3$="Anywhere, USA"
```

```
FOR i=1 TO 40
 zero$=zero$+"0"
NEXT i
```

```
WIDTH 40,20
TEXTFONT(4):TEXTSIZE(12) 'Monaco
GOSUB printit
TEXTFONT(2):TEXTSIZE(12) 'New York
GOSUB printit
END
```

```
printit:
PRINT l1$,r1$
PRINT l2$,r2$
PRINT l3$,R3$
PRINT zero$
```

```
PRINT l1$;TAB(21);r1$
PRINT l2$;TAB(21);r2$
PRINT l3$;TAB(21);R3$
PRINT zero$
RETURN
```

In the sample program it is clear that the spacing of "O" is different for the 2 fonts, which explains why the second column does not line up. **RULE 1:** Remember that when mixing fonts you can't expect them to line up. This program

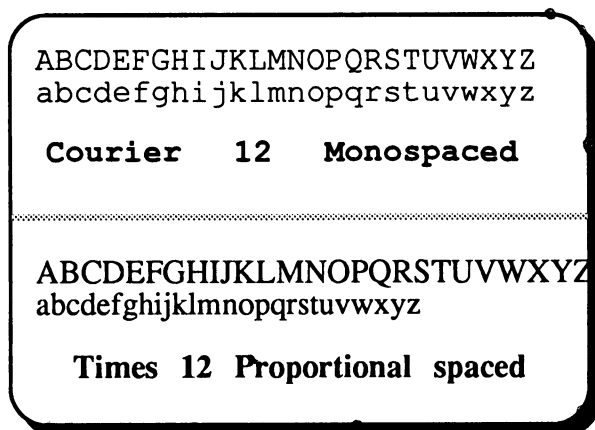


Figure 1

demonstrates the use of the BASIC WIDTH command. The syntax of the WIDTH command is defined on page 264 of the MSBASIC manual. In general the syntax is: WIDTH [size] [,print-zone] . The size is the number of standard characters that a line may contain. In other words, BASIC will force a carriage return after the line size is full. If the size is 255 then BASIC will not force any carriage returns. The print-zone assigns the width of the print zone. The print zones are similar to tab stops, and they are forced by comma delimiters in the PRINT or LPRINT statements. In the sample program, you can see that by using TAB or by using WIDTH the same result can be achieved. Using WIDTH the position of the 2nd column is determined by the print-zone. By using TAB(I), print position moves to the position indicated by I. An understanding of how these commands may be used is helpful when you are trying to format your printed output. An example would be the printing of a multiple column roster or newsletter where different fields might be combined together to create an entire line. The traditional way to do this would be to combine the fields together as one string variable and then to print the variable. The spacing of each field is filled to the size of the print-zone before combining all the fields, so that the total width of all the fields will fit. This works fine with monospaced fonts but it is quite difficult to accomplish with proportional spaced fonts. An alternative solution is to use TAB or WIDTH functions to specify where each field will be printed. It is still important to check the length of each field before printing to make sure that each field will fit into the specified print zone. If it won't fit then either the field must be truncated or the print-zone should be adjusted. If not, then the field will print past the assigned print-zone and into the next print-zone. The next field to be printed will be pushed over to the next available print-zone.

This can be demonstrated by changing the sample program. In the program the print-zone is set at 20 characters wide. If fewer than 20 characters are printed, then the next print field will begin 20 characters after the start of the first print field. Try changing the line: l1\$="Name #1":r1\$="Name #2" to a field that has more than 20 characters, for example: l1\$="Name #1 is too long now":r1\$="Name #2" to demonstrate the problem. To fix the line change it to:

```
l1$=LEFT$("Name #1 is too long now",20):r1$="Name #2"
or adjust the WIDTH statement to: WIDTH 40,25.
```

Another consideration when using fonts of different sizes and widths is that each font will use a different number of lines to print the same thing. Because there are so many different fonts and sizes available it is difficult to examine the effect of each one. So before using any set of fonts, you should test each one out to see how many lines will fit on a page, and adjust your program to count the lines as they are printed. If you are printing in a 10-point font, it is probable that you will have more lines than for a 12-point font. The Disk Labeler program is a quick sample of what might be done with the WIDTH statement. Keep in mind that this is just a

sample and may or may not be a useful application, however it may be modified to meet your needs.

### Disk Labeler Program Sample

The program will read filenames one at a time from the disk and place them in an array called filename\$(i). I have limited the size to 10 files to simplify the program. Choose Add/Remove from the files menu to add or remove a file. The selected files are printed in window 1. The second menu lets you select the font info. This illustrates a method of changing the fonts and sizes that will be used. This could be expanded to select various attributes such as Bold or Italics. Keep in mind that the different attributes will also change the width of each character. The printlabel routine demonstrates printing different fonts to the printer. If you have had trouble changing fonts when printing to the printer you may want to take a closer look at this routine. *The key is the WINDOW OUTPUT #1 statement.* This statement is necessary to direct all graphics commands, including those which change the fonts, to the printer. The Set Disk Label item of the files menu will let you set the number of lines and the number of columns that will be printed. The columns are formatted with the WIDTH statement. Disk labels may be printed in any desired font by adding additional fonts to the menu. I recommend printing labels in a small font (9 point) in order to fit more on a label. **Have any useful printing tips? Send your best suggestions to me care of MacTutor!**

```
' Disk Labeler
' By Dave Kelly
' MACTUTOR ©1985
```

```
' Watch line breaks. A few Basic lines are longer than
' this Journals column width, but the breaks are
' obviously placed so that it is clear the Basic line
' continues from the previous line.
```

```
number.of.files=0:DEFINT i-m:maxfiles=10
number.of.lines=5:number.of.col=2
DIM filename$(maxfiles)
```

```
' fonts
chicago=0: Appl=1: Newyork=2: Geneva=3: Monaco=4
activefont=Geneva: activesize=12
WINDOW 1,"",(200,50)-(400,275),2
'menu clear
FOR i=1 TO 5:MENUS i,0,0,"":NEXT i
```

```
'set new menus
```

```
MENU 1,0,1,"File"
MENU 1,1,1,"Add/Remove a file"
MENU 1,2,1,"Clear all files"
MENU 1,3,0,""
MENU 1,4,1,"Print Disk Label"
MENU 1,5,1,"Set Disk Label"
```

**MENU 1,6,1,"Quit"**

**MENU 2,0,1,"Font Info"**

**MENU 2,1,0,"Fonts"**

**MENU 2,2,1,"NewYork"**

**MENU 2,3,2,"Geneva"**

**MENU 2,4,1,"Monaco"**

**MENU 2,5,0,"-"**

**MENU 2,6,1,"9 point"**

**MENU 2,7,1,"10 point"**

**MENU 2,8,2,"12 point"**

**ON MENU GOSUB Menucheck:MENU ON**  
**idle: GOTO idle**

Menucheck:

Menunumber=**MENU(0)**: Menuitem=**MENU(1)**  
**ON** Menunumber **GOSUB** Filemenu, Fontmenu  
**RETURN**

Filemenu:

**MENU**

**ON** Menuitem **GOSUB** Add, Clearfiles, Blank, Printlabel,  
Setlabel, Quit  
**RETURN**

Add:

x\$=**FILES\$(1)**

**IF** x\$="" **THEN GOSUB** display:**RETURN**

'x\$=**MID\$(x\$,INSTR(x\$,"")+1)**

'optional line:use to strip volume name

**FOR** i= 1 **TO** number.of.files

**IF** x\$=filename\$(i) **THEN** filename\$(i)="zzzzz":

i=number.of.files+2: **CALL** Sort(filename\$(i),  
number.of.files): number.of.files= number.of.files-1

**NEXT** i

**IF** number.of.files>=maxfiles **THEN GOSUB** display:  
**RETURN**

**IF** i=number.of.files+1 **THEN** filename\$(i)=x\$:  
number.of.files=i

**CALL** Sort (filename\$(i),number.of.files )

**GOSUB** display

**RETURN**

Clearfiles:

number.of.files=0

**GOSUB** display

**RETURN**

display:

**TEXTFONT**(chicago):**TEXTSIZE**(12)

**CLS:PRINT** number.of.files;"files have been selected."

**TEXTFONT**(activefont):**TEXTSIZE**(activesize)

**FOR** i=1 **TO** number.of.files

maxlength=30

**IF** activesize<=10 **THEN** maxlength=40

**PRINT** **LEFT\$(filename\$(i),maxlength)**

**NEXT** i

**IF** number.of.files=maxfiles **THEN**

**PRINT**"Max.";maxfiles;"files allowed"  
**RETURN**

Blank:**RETURN**

Printlabel:

**IF** number.of.files=0 **THEN RETURN**

**OPEN "LPT1:PROMPT" FOR OUTPUT AS #1**

**WINDOW OUTPUT #1**

**TEXTFONT**(activefont):**TEXTSIZE**(activesize)

colwid=30:**IF** number.of.col=2 **THEN** colwid=20

**WIDTH "LPT1:"**, (colwid+1)\*2,colwid+1

**PRINT**: items=number.of.files

**ON ERROR GOTO** Error.handle 'check for printing abort

**IF** number.of.col=1 **THEN** onecol

**IF** number.of.lines<number.of.files/2 **THEN**

items=number.of.lines\*2

**FOR** i= 1 **TO** items **STEP** 2

**PRINT** #1,**LEFT\$(filename\$(i),colwid)**,  
**LEFT\$(filename\$(i+1),colwid)**

**NEXT** i

**CLOSE** #1

Exit.printing:

**WINDOW OUTPUT 1**

**GOSUB** display

**RETURN**

onecol:

**IF** number.of.lines<number.of.files **THEN**

items=number.of.lines

**FOR** i= 1 **TO** items

**PRINT** #1,**LEFT\$(filename\$(i),colwid)**

**NEXT** i

**CLOSE** #1

**GOTO** Exit.printing

Error.handle: **RESET:RESUME**

Exit.printing:**STOP**

Setlabel:

**CLS:TEXTFONT**(0):**TEXTSIZE**(12)

**LOCATE** 1,1:**PRINT**"Lines per label:"

**LOCATE** 3,1:**PRINT** "# of columns:"

lines\$=**STR\$(number.of.lines)**

cwid\$=**STR\$(number.of.col)**

**EDIT FIELD** 2,cwid\$,(125,31)-(155,46),1,2

**EDIT FIELD** 1,lines\$,(125,1)-(145,16),1,2

**BUTTON** 1,1,"OK",(25,70)-(75,88)

i=1

loop:

d=**DIALOG**(0)

**IF** d=1 **THEN** Done ' got OK button

**IF** d=2 **THEN** i=**DIALOG**(2):**EDIT FIELD** i 'got field

**IF** d=6 **THEN** Done 'got return key

**IF** d=7 **THEN** i=(i **MOD** 2)+1:**EDIT FIELD** i 'got TAB key

## **GOTO loop**

```
Done:
lines$=EDIT$(1)
number.of.lines=VAL(lines$)
IF number.of.lines<1 OR number.of.lines>9 THEN loop
 cwid$=EDIT$(2)
 number.of.col=VAL(cwid$)
 IF number.of.col<1 OR number.of.col>2 THEN loop
 EDIT FIELD CLOSE 1:EDIT FIELD CLOSE 2
 BUTTON CLOSE 1
 CLS
 GOSUB display
 RETURN
```

```
Quit:
 MENU RESET
 WINDOW 1,"Disk Labeler",(2,40)-(510,250),1
 END
```

```
Fontmenu:
ON Menuitem GOSUB Blank,NY,Geneva,Monaco,
 Blank,Pt09,Pt10,Pt12
GOSUB display
RETURN
```

```
NY: activefont=Newyork
MENU 2,2,2: MENU 2,3,1: MENU 2,4,1
RETURN
```

```
Geneva: activefont=Geneva
MENU 2,2,1: MENU 2,3,2: MENU 2,4,1
RETURN
```

```
Monaco: activefont=Monaco
MENU 2,2,1: MENU 2,3,1: MENU 2,4,2
RETURN
```

```
Pt09: activesize=9
MENU 2,6,2: MENU 2,7,1: MENU 2,8,1
RETURN
```

```
Pt10: activesize=10
MENU 2,6,1: MENU 2,7,2: MENU 2,8,1
RETURN
```

```
Pt12: activesize=12
MENU 2,6,1: MENU 2,7,1: MENU 2,8,2
RETURN
```

```
SUB Sort (item$(maxfiles),N) STATIC
FOR i=1 TO N-1
 FOR j=i+1 TO N
 IF item$(i)> item$(j) THEN SWAP item$(i),item$(j)
 NEXT j
NEXT i
END SUB
```



# Basic School

## Building A Library Routine To Eject The Disk



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 12

This month's BASIC column features some of the ins and outs of installing your own 68000 code to your MS BASIC programs. The potential power this provides is mind-boggling. With just a few lines of code you can access some of the other ROM routines that are not provided with MS BASIC 2.0.

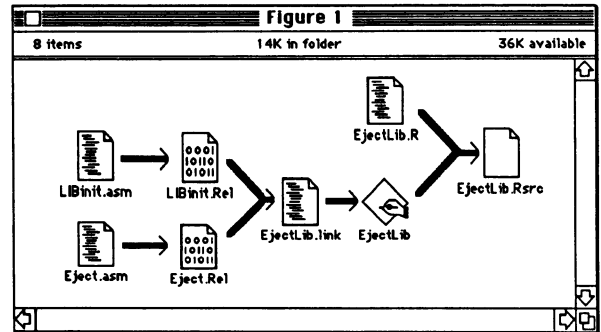
First off, if you are not already familiar with the Apple Macintosh 68000 Development System, then I suggest that you read through the manual to become familiar with the system. In MacTutor Vol. 1 No. 1, the Assembly Language Lab column provides an overview of the MDS system that will be useful for our discussion in this column. It is not necessary that you know 68000 assembly to enter and compile the BASIC Library we will be installing. I will try to lead you through step by step from the point of view of someone that is not very familiar with the MDS system.

Now before we get into the thick of things, we will discuss the BASIC **LIBRARY** statement. The **LIBRARY** command is a BASIC reserved word but there is no description of it in the BASIC manual other than a brief mention of it on page 103. This page also refers to some special documentation entitled "Microsoft BASIC for the Macintosh -- Building Machine Language Libraries" (hereafter abbreviated as BMLL) which is available by contacting the Microsoft Consumer Response Department. The BMLL document has all the necessary information required to set up your own machine language library and includes 3 sample libraries: CopyFile, PrintArgs, and AddStrings.

A BASIC Library is a Macintosh resource with one or more named CODE segments. Refer to the Inside Macintosh Programmer's Guide section for the definition of the file structure of a resource file. Other sections of Inside Macintosh will be useful in setting up your own routines.

To open a library file you use the statement **LIBRARY** <filename string expression>. Up to 5 library files may be opened at the same time. When the Library file is opened, BASIC calls the routine LIBinit which must exist in each file. The purpose of LIBinit is to allow a way for you to inform BASIC of the compatibility of your library routine with the version of BASIC being used. If the library routine is incompatible or the LIBinit routine does not exist or the handshake with that routine fails, then the library is closed and an "Illegal function call" error is generated. This statement can result in other errors which are described in BMLL.

Once the library has been opened, the routines within the library may be called by BASIC with the BASIC statements: <routine name> [argument-list] or by **CALL** <routine name>



Constructing a library resource for BASIC

[(argument-list)]. More on the **CALL** statement may be found in your BASIC manual starting on page 101.

A library is detached from BASIC whenever **NEW**, **SYSTEM**, **RUN**, or **LIBRARY CLOSE** is executed. To close the library, BASIC looks for the routine called LIBterm if present and executes it. The routine LIBterm should release any memory which was allocated from the heap and close any files opened by routines in the library. Unless you need to allocate or deallocate memory you probably won't need to use LIBterm.

A sample source file for LIBinit is given with BMLL, but you'll have to work out your own LIBterm if needed, depending on the routine you are writing. The sample LIBinit file demonstrates how the routine can check the BASIC version and see if it is compatible with the routines in the library. This sample checks to see that the binary version of BASIC is being used and tells BASIC if it is not compatible. Also included is an equate file named BASIC.D which defines entry points in BASIC which you should use to communicate with BASIC. For my sample library I have included all applicable equates in my source listing, so the equate file will not be needed. You will want to use these routines to allocate memory and read the variable argument lists in the BASIC **CALL** statement. It is advisable that you obtain BMLL before writing your own routines, as there are a lot of useful routines that can be used that are part of BASIC. BMLL explains what each routine does and what registers are affected.

Alright, now we'll get right down to business. First we need to create several files with the MDS editor which are to be used by the assembler, linker and resource compiler. The two code segments that we want to create are listed below and should be saved with the file names shown. The LIBinit routine will be installed as the first segment and our routine called Eject will be the second segment. Eject will allow you to eject either the internal or external disk from within your

program (under BASIC control). Type in the two routines then save them to disk.

Save this routine as "LIBinit.asm".

```
;
; LIBinit ROUTINE
; Sample version of LIBinit routine
;
; REGISTER INFORMATION
;
; A0 = pointer to a version record containing:
; 2 bytes = version of Basic interpreter (ie 2 for 2.01)
; 2 bytes = revision number of Basic interpreter (01)
; 2 bytes = 0 if decimal math, 1 if IEEE binary format
; 2 bytes = compatibility variable of this routine to Basic:
; 0 = compatible
; -1 = incompatible
; 8 bytes = reserved array of four INTEGER values
;
; A4 = pointer to a handle (long word) owned by this library
; Use this as a handle to a static data segment.
; A5 = pointer to the base of the application jump table
; D0 = 0 on exit if this routine is purgeable.

LIBVER_Result EQU 6 ;offset to version record field
;for compatibility. (See above)

LIBinit:
 CLR.W LIBVER_Result(A0) ;assume compatible

LIBinitExit:
 MOVEQ #0,D0
 RTS
```

Here is the Eject Library source code. Save this file as "Eject.asm".

```
; BASIC Eject Library Source Code
; By Dave Kelly
; MacTutor 1985
```

```
;Synopsis:
; CALL Eject (VolRefNum)
;Output:
; The specified disk is ejected (not dismounted) from
; the selected drive where VolRefNum may be 0 for the
; default drive, 1 for the internal drive, 2 for the
; external drive.
```

```
GetNextLibArg EQU $2A ; Basic Lib offset from A5
IntegerArg EQU $32 ; Basic Lib offset from A5
BasicError EQU $42 ; Basic Lib offset from A5
```

```
.TRAP _Eject $A000+23 ;toolbox trap
```

```
Ejectdisk:
 BSR GetIntegerVar ;integer arg in [d3:w]
```

```
 CMP.W #0,d3
 BEQ Doit ;branch if on 0 (default drive)
 CMP.W #1,d3
 BEQ Doit ;branch if on 1 (internal drive)
 CMP.W #2,d3
 BEQ Doit ;branch if on 2 (external drive)

UnknownVol:
 MOVEQ #74,d2 ;Unknown Volume error
 JSR BasicError(a5)

Doit:
 LEA ParamBlock,a0 ;get pointer
 MOVE.W d3,22(a0) ;move drive # to ioVRefNum
 _Eject
 RTS
GetIntegerVar:
 JSR GetNextLibArg(a5) ;Get the next
;argument
 JSR IntegerArg(a5) ;[d3:w] = integer
;{error if arg can't be forced
;into an integer

 RTS

; local data area

ParamBlock:
 DC.L 0 ;ioLinkptr.
 DC.W 0 ;ioType
 DC.W 0 ;ioTrap
 DC.L 0 ;ioCmdAddr
 DC.L 0 ;ioCompletionptr.
 DC.W 0 ;ioResult
 DC.L 0 ;ioFileNameptr.
 DC.W 0 ;ioDrvNum (ioVRefNum)

 END
```

By the way, the Eject routine shows an example of using the file manager parameter block similar to what has been discussed in a few places in the last few MacTutor issues. Refer to those columns for more information. (ie Vol. 1 number 7 & 8)

Next we need to start a new MDS editor file to create a link file for the linker to use. The link shown below should be typed in and saved as "EjectLib.link". Note that if you are not using the default (startup) disk to save these files, the volume name must also be included with the filename.

```
;Eject Library Link
;MacTutor 1985
```

```
/OUTPUT EjectLib
LIBinit.Rel
<
Eject.Rel
$
```

Now create one last editor file for the Resource compiler. This file creates the resource file that BASIC will call to use the library. Be sure that each line (including the last one) has a carriage return at the end of the line or all the resources may

not be compiled. Save this file as "EjectLib.R"

```
EjectLib.Rsrc
BLIB
TYPE CODE = GNRL
LIBinit,1
.R
EjectLib CODE 1
```

```
TYPE CODE = GNRL
Eject,2
.R
EjectLib CODE 2
```

Ok, we are now ready to assemble our code segments. Assemble LIBinit.asm and Eject.asm by selecting ASM from the MDS system transfer menu, or by double clicking on the assembler if you are running the finder desktop. A standard file dialog box will appear and you can select each file and assemble them one at a time. Two new files containing the assembled code are stored on the disk, LIBinit.Rel and Eject.Rel. These two files will be used by the linker to create a linked application file. Select the LINK application from the transfer menu of the MDS system. Note you may have to push the cancel button from the dialog box to get to the transfer menu. The LINKER will create an application to be used by RMAKER. Don't try to run the EjectLib application as it will BOMB!! It only contains resource code segments and is not a regular application program. After linking, select RMAKER from the transfer menu and use the EjectLib.R file to create the resource file to be used by our BASIC library.

Figure 1 shows the files required and the flow of the above procedures. If you have any problems with this then study the MDS system manual and the BMLL supplement.

To try out the Eject Library routine, type in the Eject Demo program. To call the Eject routine, open the Eject Library and then use CALL Eject (VolRefNum). The VolRefNum is a number for the volume which you want to eject. A "0" will eject the disk in the default drive, a "1" will eject the internal drive and "2" will eject the external drive. For any other VolRefNum the file manager returns a unknown volume name error which could be trapped in your BASIC error trap routine if necessary. By the way, using ResEdit you can move the resource segments from the resource files into your BASIC program file (data file where your program is stored). Then the routine can be called by opening up the library with the filename the same as the program. The program would now be completely transportable as one file.

Installing the routines is the easy part. After trying out the demo routines in the BMLL supplement and the Eject routine, your mouth may be watering for some more. Well, next time we will take a look at some routines which have been pre-written to attach to your BASIC programs allowing access to over a hundred more ROM routines. They are available from a company called Clear Lake Research, 5353 Dora Street #7, Houston, Texas 77005 (1-800-835-2246 X199). If you've felt that BASIC just didn't have enough access to ROM routines, then check this one out.

'Eject Library Demo  
'By Dave Kelly  
'MacTutor ©1985

'Note: EjectLib.Rsrc must be on default disk  
'or you must specify vol name in Library statement  
**LIBRARY "EjectLib.Rsrc"**

```
'Set up menus
FOR i=1 TO 5
 MENU i,0,0,""
NEXT i
MENU 1,0,1,"File"
MENU 1,1,1,"Quit"
MENU 2,0,1,"Eject Disk"
MENU 2,1,1,"Eject Default Disk"
MENU 2,2,1,"Eject Internal Disk"
MENU 2,3,1,"Eject External Disk"
```

**ON MENU GOSUB Menucheck:MENU ON**  
**pause:GOTO pause**

```
Menucheck:
 menunumber = MENU(0)
 menuitem=MENU(1):MENU
 IF menunumber = 1 THEN filemenu
 IF menunumber <>2 THEN RETURN
 IF menuitem = 1 THEN vol= 0
 IF menuitem = 2 THEN vol = 1
 IF menuitem = 3 THEN vol = 2
 CALL Eject (vol)
 RETURN
```

```
filemenu:
 IF menuitem <> 1 THEN RETURN
 MENU RESET:LIBRARY CLOSE:END
```

Thanks go to Robert Millis for sending us an improvement to the Cursor Editor routine in the August '85 issue. Due to editorial deadlines it is not possible to streamline all the code used in each issue for maximum efficiency. We do try to make sure that each and every program works. Keep in mind also that the code for most of the columns must be edited to fit properly. The programs are run before editing and placed on the source disks which you may order from us. We encourage you to modify and improve our programs. Please feel encouraged to send us any improvements that you have made so that our other readers may also benefit. The following code may replace the getpixel routine in the August Cursor Editor for improved speed and useability of the program:

```
getpixel: 'For Cursor Editor
 pixel%=256
 xp=MOUSE(1):yp=MOUSE(2)
 IF xp<20 OR xp>196 THEN RETURN
 IF yp<20 OR yp>196 THEN RETURN
 row=INT((yp-20)/11):col = INT((xp-20)/11)
 pixel%=row*16+(15-col)
 RETURN
```

It should be noted that the same routine was used in the September Paint Pattern Editor with some minor differences.



Don't mix them up; they are different. The following code may replace the getpixel routine in Paint Pattern Editor:

```
getpixel: 'For Paint Pattern Editor
 pixel%=64
 xp=MOUSE(1):yp=MOUSE(2)
```

```
IF xp<20 OR xp>108 THEN RETURN
IF yp<20 OR yp>108 THEN RETURN
row=INT((yp-20)/11):col = INT((xp-20)/11)
pixel%=row*8+(7-col)
RETURN
```



# Basic School

## Play Icon Concentration



Dave Kelly  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 13

Welcome to BASIC School, Happy Birthday MacTutor!! and Merry Christmas. This month features a way to use icons from within your programs.

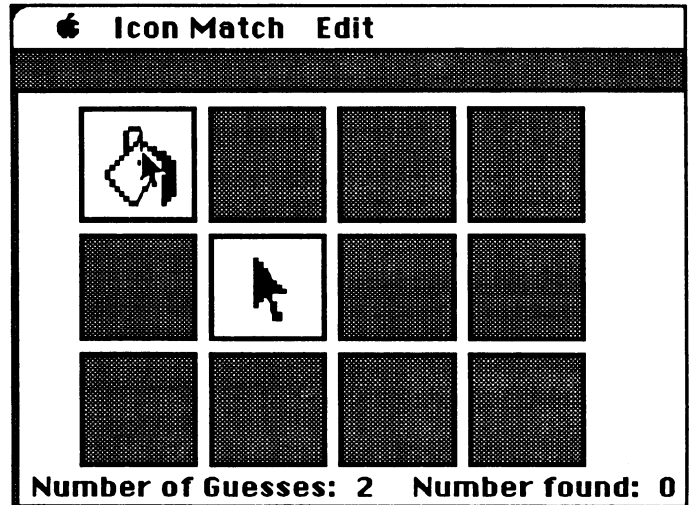
The Macintosh's greatest strength is the ability to communicate graphically with the user. Like they say, "just point and click", the user has very little need to know the inner workings of the software. If you are developing software for other people to use, you will want to make things as easy as possible and as "bullet proof" as possible. If you are programming for users who have little aptitude for computers, you may want to use the Mac graphics abilities to communicate to the user what to do.

In investigating the use of the Mac for our manufacturing line at Hughes Aircraft, a demonstration program was developed which allowed the user to input serial numbers via a bar code reader, and then by pointing and clicking at icons on the screen the user could select the various failure modes associated with the manufacturing process. One icon might be a smiling or frowning face indicating a passing or failing operation. The first task was to read pictures (icons) created in MacPaint or MacDraw and store them into files on the disk which the programs could use as a 'resource'. Note I use the word resource very loosely here. The icons are not actually stored in Macintosh resource files, but are stored in an ordinary sequential data file. There are library resources available from Clear Lake Research which can store icons in resource files for later use.

The Icon Match Game program actually contains two programs in one. The first part of the listing below contains the game itself. The second half contains the icon file creator utility which may also be used to create icon files that you may use in your own programs. The icon's creator program (see below) reads the icon from the clipboard (see "The Art of Clipboarding", MacTutor May 1985) and stores an array of icons in a file on the disk. The icons may then be read from the file when needed and displayed on the screen.

### Creating The Icon Files

The first step is to decide which pictures you want to use. The pictures can be anything of type PICT which can be stored in the scrapbook. Create 12 pictures and store them in the scrapbook. The pictures may be any size as the picture is scaled after it is read from the clipboard. Be sure that the scrapbook where you save your pictures is on the default disk when you run the program. If the desired scrapbook is on the



same disk as BASIC or you have a desk accessory to change the default drive you will be able to access the pictures you created.

Run the program and select 'CREATE New Icons' from the menu. A new menu comes up which allows you to add icons to the file. Now open up the scrapbook and copy the desired picture (icon) to the clipboard. Instead of closing the scrapbook, click on the output window then select 'Add an Icon' from the 'Add Match Icons' menu. The current picture is read from the clipboard and displayed on the screen using the PICTURE statement. The picture on the screen is then read using a screen GET command and stored in an array. It is this array which is stored on the disk for later retrieval. The GET statement is used for two reasons: 1. using GET and PUT is a much faster way to display graphics on the screen. 2. Several icons may be stored in a single file, instead of using several small files to store each icon, one per file. The program demonstrates the use of multidimensional arrays in GET and PUT statements. Note that the picture is stored in the first dimension of the array and any additional dimensions are added after that.

To add additional pictures, click on the scrapbook window to select the scrapbook again and copy the next picture to the clipboard. Then click on the output window and add the icon to the program file. If you don't select the output window before adding a new icon, the GET command may not get the whole icon if the scrapbook is partially covering it when it displays on the screen. If you make a mistake you can clear

the icons and start over. After 12 icons have been entered you may save them to the disk using the 'Save Current Icons to Disk' command or return to the Icon Match game to use the icons in the game. On my own disk, I have created different icons files for my sons to play with. One contains the first 12 letters of the alphabet for an alphabet match game. Another file contains icons that are used in MacPaint, the lasso, the hand, the pencil etc. [The icon file is available with the source code on the source code disk. -Ed.]

### Icon Match Game

The icon match game uses the icons created with the icon file creator. If a set of icons is already loaded the match game uses the current set, otherwise you are prompted for the filename of an icon file. The arbitrary file type 'BICN' was invented to distinguish an icon file from other types of files. After the file is loaded if necessary, the icons are shuffled and 24 boxes are drawn on the screen. Clicking on a box shows the icon hidden behind the box. Clicking on a second box shows another icon. The object is to find all 12 pairs of icons by opening up the boxes and matching the pairs together. Unless you are lucky, you will have to use your memory to remember the positions of the icons to match them together. The boxes will disappear as icons are matched, indicating that the pairs or icons have been found. Your score is displayed at the bottom of the screen. Have fun.

'Icon Match Game  
'By Dave Kelly  
'MACTUTOR ©1985

'Main Program initialization  
WINDOW CLOSE 1  
DEFINT a-z 'set variables as type integer  
iconsize%=500  
DIM length(12), icon\$(12), iconput%(iconsize%,12)  
DIM r%(24), door(24)  
FOR i= 0 TO 3:patt%(i)=&H55AA:NEXT i 'set up pattern  
true=-1:false=0:opened=-  
1:closed=0:found=1:iconloaded=false

'set up menus  
FOR i=3 TO 5: MENU i,0,0,"": NEXT i  
MENU 1,0,1,"Icon Match"  
MENU 1,1,0,"Start Game (Current Icons)"  
MENU 1,2,1,"New Game (New Icons)"  
MENU 1,3,0,"-"  
MENU 1,4,1,"CREATE New Icons"  
MENU 1,5,0,"-"  
MENU 1,6,1,"Quit"  
setmenu:ON MENU GOSUB Mcheck:MENU ON  
endlessloop:  
IF iconloaded THEN MENU 1,1,1 ELSE MENU  
1,1,0  
GOTO endlessloop

Mcheck: 'check item for menu 1  
menunum=MENU(0):IF menunum <> 1 THEN  
MENU:RETURN  
menuitem=MENU(1):MENU  
ON menuitem GOSUB Startgame, Newgame,,  
Createicons,, Quit  
RETURN

Newgame:  
iconloaded=false  
Startgame:  
IF iconloaded=false THEN GOSUB Readicons  
IF filename\$="" THEN RETURN  
MENU 1,1,0:MENU 1,2,0:MENU 1,4,0  
numberfound=0:guesses=0  
GOSUB Shuffle 'shuffle icons  
WINDOW 1,,(2,40)-(510,275),3  
MENU ON  
FOR i=1 TO 24  
door(i)=closed  
NEXT i  
GOSUB Play  
WINDOW CLOSE 1  
MENU 1,1,1:MENU 1,2,1:MENU 1,4,1  
RETURN

Drawstatus: 'Draw the screen with current status  
i=1  
x1=30:y1=30:x2=50+y1:y2=50+y1  
WHILE i<=24  
rect%(0)=y1:rect%(1)=x1:rect%(2)=y2:rect%(3)=x2  
IF door(i)=found THEN  
ERASERECT(VARPTR(rect%(0))):  
GOTO skip  
IF door(i)=closed THEN CALL  
FILLRECT(VARPTR(rect%(0)),  
VARPTR(patt%(0))): CALL  
FRAMERECT(VARPTR(rect%(0)))  
IF door(i)=opened THEN  
ERASERECT(VARPTR(rect%(0))):  
PUT(x1,y1)-(x2-1,y2-1),iconput%(0,r%(i))  
skip:x1=x1+55:x2=x2+55:i=i+1  
IF i=9 THEN x1=30:y1=y2+5:x2=50+y1:y2=50+y1  
IF i=17 THEN x1=30:y1=y2+5:x2=50+y1:y2=50+y1  
WEND  
RETURN

Play:  
FOR i=1 TO 24  
IF door(i)=opened THEN door(i)=closed  
NEXT i  
mousee=MOUSE(0)  
GOSUB Drawstatus

Open1stdoor:  
mousee=MOUSE(0):IF mousee=0 THEN Open1stdoor  
GOSUB finddoor  
IF doornum=0 OR door(doornum)=found THEN  
Open1stdoor  
door(doornum)=opened:doorA=doornum  
GOSUB Drawstatus  
mousee=MOUSE(0)

```

Open2nddoor:
 mousee=MOUSE(0):IF mousee=0 THEN Open2nddoor
 GOSUB finddoor
 IF doornum=0 OR door(doornum)=found OR
doornum=doorA THEN Open2nddoor
 door(doornum)=opened:doorB=doornum
 GOSUB Drawstatus
 IF r%(doorA)=r%(doorB) THEN door(doorA)=found:
 door(doorB)=found:
 numberfound=numberfound+1
 guesses=guesses+1:CALL TEXTFONT(0)
 CALL MOVETO(100,210):PRINT "Number of Guesses:";
 guesses;" Number found:";numberfound
 FOR i=1 TO 2000:NEXT i
 IF numberfound<12 THEN Play
RETURN

finddoor:
doornum=0:row1=0:row2=8:row3=16
xpos=MOUSE(1):ypos=MOUSE(2)
IF xpos>465 OR xpos <30 OR ypos <30 OR ypos >190
THEN RETURN

rows:"Find selected row
IF ypos>=30 AND ypos <=80 THEN
 doornum=doornum+row1: GOTO cols
IF ypos>=85 AND ypos <=135 THEN
 doornum=doornum+row2: GOTO cols
IF ypos>=140 AND ypos<=190 THEN
 doornum=doornum+row3: GOTO cols
RETURN
cols:"Find selected column
IF xpos>=30 AND xpos<=80 THEN
 doornum=doornum+1:RETURN
IF xpos>=85 AND xpos<=135 THEN
 doornum=doornum+2:RETURN
IF xpos>=140 AND xpos<=190 THEN
 doornum=doornum+3:RETURN
IF xpos>=195 AND xpos<=245 THEN
 doornum=doornum+4:RETURN
IF xpos>=250 AND xpos<=300 THEN
 doornum=doornum+5:RETURN
IF xpos>=305 AND xpos<=355 THEN
 doornum=doornum+6:RETURN
IF xpos>=360 AND xpos<=410 THEN
 doornum=doornum+7:RETURN
IF xpos>=415 AND xpos<=465 THEN
 doornum=doornum+8:RETURN
doornum=0
RETURN

Shuffle:
WINDOW 2,,(150,100)-(350,150),-2
TEXTFONT(0):LOCATE 1,3:PRINT "Please wait...."
LOCATE 2,3:PRINT"Now shuffling icons."
RANDOMIZE TIMER
r%(1)=1:r%(2)=2:r%(3)=3:r%(4)=4:r%(5)=5:r%(6)=6
r%(7)=7:r%(8)=8:r%(9)=9:r%(10)=10:r%(11)=11:r%(12)=12
r%(13)=1:r%(14)=2:r%(15)=3:r%(16)=4:r%(17)=5:r%(18)=6
r%(19)=7:r%(20)=8:r%(21)=9:r%(22)=10:r%(23)=11:r%(24)=
12
mixstart=1:mixend=12:GOSUB mixup

```

```

mixstart=13:mixend=24:GOSUB mixup
FOR i=1 TO 12:SWAP r%(i),r%(i+6):NEXT i
WINDOW CLOSE 2
RETURN

mixup:
 FOR i = mixstart TO mixend
 getnewrnd:
 r%(i)=INT(RND*12.4):IF r%(i)= 0 THEN getnewrnd
 getanother=0
 FOR j=mixstart TO i-1
 IF r(i)=r(j) THEN getanother=1
 NEXT j
 IF getanother=1 THEN GOTO getnewrnd
 NEXT i
RETURN

Quit:
 CLS:MENU RESET:END

Createicons:
'Match Game Icon Creator
'
'Make your Icon in MacDraw or MacPaint
'then copy to scrapbook. This program will
'read your Icon from the clipboard.
'CAUTION: click once on the output window before copying
'if you don't want to close scrapbook for
'multiple clips. Use Add to add an icon to the file.
'Pictures of any size may be used. They will be scaled
'when clipped from the clipboard.
MENU 1,0,0
MENU 3,0,1,"Add Match Icons"
MENU 3,1,1,"Display Current Icons"
MENU 3,2,1,"Add an Icon"
MENU 3,3,0,"-"
MENU 3,4,1,"Save Current Icons to disk"
MENU 3,5,1,"Read Current Icons from disk"
MENU 3,6,0,"-"
MENU 3,7,1,"Clear Icons"
MENU 3,8,1,"Return to Match Game"
WINDOW 1,,(2,40)-(510,275),3

ON MENU GOSUB Checkmenu:MENUE ON
InfiniteLoop:
 IF iconloaded=false THEN MENU 3,4,0:MENU 3,2,1
 IF iconloaded=true THEN MENU 3,4,1:MENU 3,2,0
 GOTO InfiniteLoop

Checkmenu: 'check items FOR MENU 3
 menunum=MENUE(0): IF menunum<3 THEN RETURN
 menuitem=MENUE(1): MENU
 ON menuitem GOSUB Displayicons, Add,, SaveALL,
 Readicons,, Startover,

Quitcreate
 RETURN

Startover: 'Clear all icons
 CLS:count=0
 iconloaded=false
 RETURN

```

Add: 'add an icon to the file

count=count+1

GOSUB CopyfromClip

GOSUB Displayicons

IF count=12 THEN iconloaded=true

RETURN

CopyfromClip: 'copy picture from clipboard

OPEN "CLIP:PICTURE" FOR INPUT AS 1

length(count)=LOF(1)

Icon\$(count)=INPUT\$(length(count),1)

CLOSE 1

RETURN

Quitcreate: 'quit icon creation

WINDOW CLOSE 1

MENU 3,0,0,""

MENU 1,0,1

RETURN setmenu

SaveALL:

filename\$=FILES\$(0,"Enter Icon Filename:")

IF filename\$="" THEN RETURN

WINDOW 2,,(150,100)-(350,150),-2

TEXTFONT(0):LOCATE 1,3:PRINT "Please wait...."

LOCATE 2,3:PRINT "Now saving icons."

OPEN filename\$ FOR OUTPUT AS 1

FOR count=1 TO 12

FOR size= 0 TO Iconsize%

WRITE #1,Iconput%(size,count)

NEXT size

NEXT count

count=12

CLOSE 1

NAME filename\$ AS filename\$,"BICN"

WINDOW CLOSE 2

RETURN

Readicons:

filename\$=FILES\$(1,"BICN")

IF filename\$="" THEN RETURN

WINDOW 2,,(150,100)-(350,150),-2

TEXTFONT(0):LOCATE 1,3:PRINT "Please wait...."

LOCATE 2,3:PRINT "Now loading icons."

OPEN filename\$ FOR INPUT AS 1

x=LOF(1)

IF x=0 THEN LOCATE ,10:PRINT "No Icons

Exist":BEEP:CLOSE 1:RETURN

FOR count=1 TO 12

FOR size=0 TO Iconsize%

INPUT #1,Iconput%(size,count)

NEXT size

NEXT count

CLOSE 1

count=12

iconloaded=true

WINDOW CLOSE 2

RETURN

Displayicons:

CLS:TEXTFONT(0)

x1=30:y1=30:x2=50+x1:y2=50+y1

i=1

WHILE i<= count

IF iconloaded=true THEN PUT(x1,y1)-(x2,y2),

Iconput%(0,i) ELSE PICTURE

(x1,y1)-(x2,y2), Icon\$(i): GET

(x1,y1)-(x2,y2), Iconput%(0,i)

MOVETO (x2-x1)/2+x1,y2+20:PRINT i

x1=x1+70:x2=x2+70:i=i+1

IF i=7 THEN x1=30:y1=120:x2=50+x1:y2=50+y1

WEND

RETURN



# Special Projects

## A MacPaint Simulator in Basic



Mike Steiner  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 13

### MacPaint Functions in Basic!

The program written in Microsoft BASIC accompanying this column simulates some of the features of MacPaint. "Why," you are probably asking yourself, "would anyone want to simulate MacPaint in a BASIC program when every Macintosh comes with MacPaint at no extra cost?" The reason is that transferring a MacPaint picture to a BASIC program is not the easiest thing in the world to do. Having your own Basic version of MacPaint can be very helpful. [The real reason is the challenge! And Mike has succeeded admirably. -Ed]

Here are some of the problems of trying to mix pict's with Basic: After making the picture, you have to copy it to the clipboard (which limits you to the size of the Macpaint window, thus eliminating the ability to make a full-screen picture) before entering BASIC. Once in BASIC, you then have to transfer the picture from the clipboard to the screen. If you want to use the picture again, you have to save it as a disk file. The program with this article will let you draw a picture, save it to and load it from a disk. The routine that loads pictures from disk can be incorporated into your own programs to let you conveniently use pictures in them.

### Program Notes

The program is fairly straightforward and does not use any programming "tricks": everything used is documented in the Microsoft BASIC manual. However, it requires a 512K Macintosh. A 128K Mac does not have enough memory to handle the necessary arrays and heap requirements.

Instructions are incorporated into the program and are accessible via the Instructions option in the Options menu. There is one area where operation of the program differs from the standard and bears mentioning here. When you save a picture to disk, you are not limited to saving the entire picture. You must select the section of the picture (to include the whole picture if you want) as the first step in the save process.

*The source code to this and all the programs in MacTutor are available on Diskette. See page 71.*

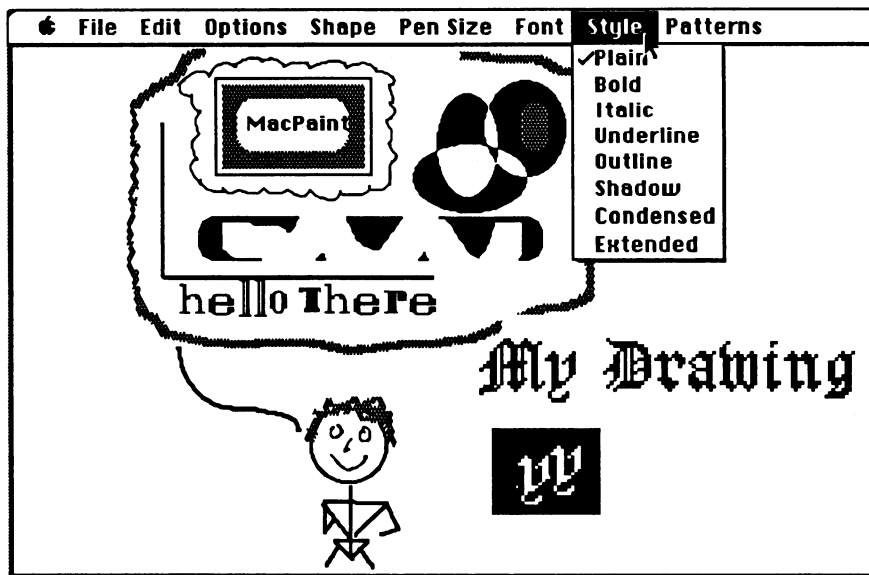


Fig. 1 Screen of MacPaint Simulator

### How It Works

The program starts by CLEARing memory from the data segment to increase the application heap memory. The heap is where BUTTON, EDIT FIELD, and PICTURE information are stored. Without the increase of heap memory the program results in a system error 25 (out of heap space) whenever it tries to print a picture.

The program then performs a GOSUB to init, which is the initialization routine. This routine causes all variables to default to integers unless they are declared otherwise. (The only other variable format used is string variables.) The program then DIMensions the arrays it will use and generates the drawing window and the program menus. It then displays the names of the program and programmer and the version and date. (This display is also available by selecting "About..." from the File menu.) It then initializes variables and the cursor arrays and RETURNs to the main program.

The routine called start is the main routine of the program. All subroutines eventually return to start. While in the start loop, the program enables drawing and checks for a menu selection. The loop sends the program to the subroutine cursorposition. This subroutine monitors the position of the cursor on the screen and changes the cursor to an arrow when it

is outside the drawing window and changes it to a cross when inside the window. Note the statement "dummy = MOUSE (0)" preceding the GOSUB. References to MOUSE (1) through MOUSE (6) return values related to the most recent click of the mouse button. MOUSE (0) checks for a mouse click, so even though it is not needed by the cursorposition routine, it must have been referenced for the routine to work. The 'ON shape GOSUB' line sends the program to the appropriate drawing routine, depending on what shape has been selected to be drawn.

### Drawing Shapes

The routines box, roundbox, and oval are essentially the same routine, differing only in the Quickdraw routines they use. They first check to see if the mouse button has been pressed. If it has not, the routines RETURN to start. If the button has been pressed, the routine enters a loop that first goes to a subroutine called getpoints. Getpoints determines the coordinates of diagonally opposite corners of a rectangle. It assigns the top left corner as the start and the bottom right corner as the end, putting the coordinates in the variable array "rect." Control is then returned to the shape-drawing routine where the shape is drawn twice and getpoints is entered again. Because the penmode was set to 10 (XOR) before the routine was entered, the pen reverses the color of the screen it passes over; by drawing the shape twice, the shape continually draws and erases itself. When the mouse button is released, control is returned to the drawing program where the pen mode is set to 8 (copy) and the shape drawn once again to leave it on the screen. The routine then RETURNs to start.

The straightline routine is similar to the framed shape routines except that starting and ending points are not swapped to insure that start is to the upper-left of finish, and a straight line is drawn. The subroutine called free is the simplest of all. It continually draws a line to the current cursor position from the previous cursor position after

moving initially to the point where the mouse button was pressed.

Text first sets the text mode to 1 (OR). A bug in the quickdraw routine that draws text puts blank spaces after the text that is printed. These spaces will erase any black pixels that are close to the end of the text. Setting the text mode to OR eliminates this problem. The routine then sets the selected text size. The first time the routine is entered after it is selected, when the mouse button is pressed, the left margin is set at the cursor's horizontal position. Text is entered normally, and when the Return key is pressed, text continues on the next line at the left margin. You may change fonts, size, and styles of the text while entering it without losing your character's position on the screen. A carriage return uses the current left margin as the left margin. To change margins, move the cursor to where you want the margin to be and click the mouse button.

### Selecting Menu Items

When a menu item is selected while the program is in the start routine's loop, the number of menu selected is stored in the variable menu0, which determines the subroutine to be entered. These subroutines are described below in increasing order of complexity.

Menu4 (Shape) checks the menu item selected and unchecks the previously selected item. It stores the number of the selected item in the variable 'shape.' Menu5 (Pensize) operates identically to Menu4, using the variable 'size' to store the pen size selected. Menu6 (Font) is similar to Menu4, but has to take into consideration that the fonts begin with zero rather than 1. Also, the program does not check to see which fonts are in the System. If a font not in the system is chosen, the default font (currently Geneva) is automatically used even though the menu indicates a different font was chosen.

Menu7 (Style), again, is similar to Menu4, but is a multi-check menu (i.e., more than one item may be checked at the same time), so the routine determines whether item selected is checked or not and toggles the check mark on or off. Also, checking menu item 1 (Plain) turns off all the other checkmarks. To further complicate this routine, TEXTFACE uses a geometrical progression to determine the style in order for the various styles to be combined. As a menu item is checked or unchecked, the variable 'face' has to have the appropriate value added or subtracted.

Menus 1 (File), 2 (Edit), and 3 (Options) enter subroutines rather than assign values to variables as do 4 through 7. Menu3 gives the choice of selecting the size of the text, using the eraser, clearing the screen, or viewing the instructions. In increasing order of complexity, these routines are as follows:

Clearscreen does a CLS and returns to start. Instructions clears the screen and displays three pages of instructions, waiting for the OK button to be pressed after each page is read.

Erase works the same way as does free, except that it uses an eraser-shaped cursor (the same one that Macpaint uses) and that the pen mode is set to 11 (BIC) which changes all black pixels to white, thereby erasing all that it passes over.

Textsize overlays a second window on the drawing window, first storing the screen area that is obliterated by the window (the way this is done is explained in the portion of this article covering Cut, Copy, and Paste). The window displays an edit field and an OK button along with instructions to type the size of the text. A WHILE — WEND loop waits for the OK button to be clicked or the Return key to be pressed. It also monitors the cursor position, changing it to an I-beam when it is in the edit field. Because the Macintosh cannot handle text sizes

smaller than four points or larger than 127 points, the routine wrongsize is used to inform the user that he has selected a size that is out of range if he should do so. After a valid size has been selected, the window is closed and the portion of window 1 that was erased by window 2 is restored.

Menu2 handles Cut, Copy, and Paste. Cut and Copy use the routine called getit. Getit goes to a subroutine called corners. This routine determines the corners of a selection rectangle the same way that the shape drawing routines use and then RETURNS to getit. Getit then GOSUBs to dimbox. Dimbox DIMENSIONS the array 'box,' using the formula given in the BASIC manual, and does a screen GET for the area selected. It then RETURNS to getit. Getit then determines whether it was called by Copy or Cut and, if the latter, erases the selected area of the screen. Dimbox is the routine that also saves the screen area behind dialog windows displayed in textsize, Open, Save, Print, and About...

Paste uses the routine putit. This routine first GOSUBs to corners. Then if corners returns a selection rectangle, it PUTs the information stored in box into the rectangle, scaling it to fit. If no rectangle was selected, it PUTs the information on the screen, unscaled, where the cursor is when corners is exited. This position is the top left corner of the pasted picture. A modified form of putit is used to restore screen information when a dialog window is closed. Note that Basic itself does not even do this when system windows overlap!

Menu1 offers the options to start a new picture, Open a picture stored to disk, Save a picture to disk, Print a picture, and get information about the program. About... opens a second window, storing the information behind it as described above, loads a picture called "About..." into the window, and waits for the OK button to be clicked. It then closes the window and restores the background. New is an alternate entry point for the clearscreen subroutine.

Open presents a dialog box that displays only those files generated by this program. The files are sequential text files of Type MIKE. After the file is selected, the window is closed and the background restored. The selected file is read into the variable 'image\$' and the PICTURE statement puts the picture on the screen in the same place and size as it was when it was saved.

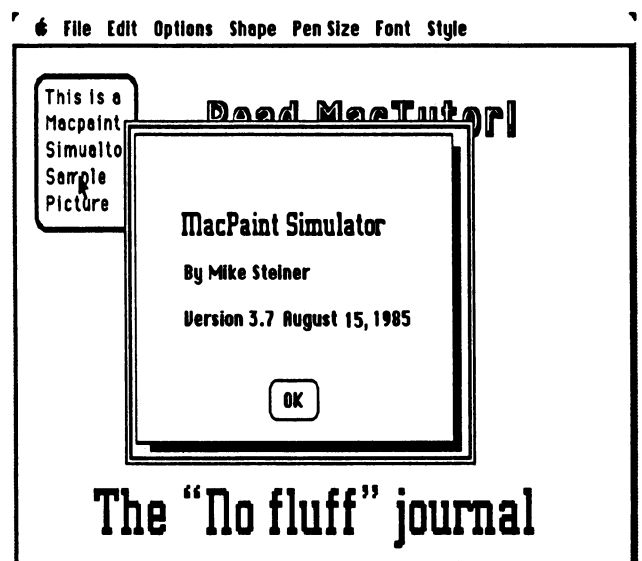
Save first waits for the user to select the portion to be saved, using getit as described above. PICTURE ON then causes any screen action to be placed in a buffer in memory. A screen PUT then places the data selected into the buffer and PICTURE OFF ends the record mode. The routine then displays a dialog box requesting the name of the file to which the picture will be saved. After Save is clicked, the window is closed, the background data restored, and the picture printed to disk. The file is then closed and the Type changed to MIKE.

Printit works very much as does Save, with the following exceptions: It prints the entire screen rather than a selected portion, and it sends the picture to the Imagewriter rather than to a disk file. By use of "LPT1:Prompt" rather than the more often used "LPT:" the program allows for selection of tall or tide orientation of the paper, normal or high quality printing, and (if you have the new Imagewriter driver file), 50 percent reduction.

More elements may be added to the program if you want to do so. A few of these are irregular shaped pens (such as | or —), filled areas, patterns for areas and lines, and polygons. I leave this as an exercise for you should you be so inclined.

## Sidebar

The MacPaint Simulator requires that the picture file "About..." be on the same disk as the program. If the program were being copied from one machine to another or being transmitted over a modem, this would be no problem. But, how do you get it to another person via the printed media, such as *MacTutor*? This problem proved to be almost as interesting as writing the program itself. The answer became obvious when I realized that the picture file is a sequential text file. I wrote a program (Read Picture) that read the file in one byte at a time, translated the bytes to their ASCII code, and wrote the ASCII as DATA statements to a file called Datafile. Once I had the DATA statements on file, I wrote a short program that translates them back into characters and writes them to a disk file called "About..." I then renamed the program "Make Picture." Both listings are provided here. You need to type in and run Write Picture before running Paint Simulator. Read Picture is provided so that if you need to send a picture file to someone by this means you will have the tools to do so. The programs are provided in their basic forms, designed only to translate "About..." If you plan to use them often, you will probably want to modify them to prompt you for the names of the respective files.





**CLEAR,250000!**

**GOSUB init**

**start:**

**IF shape <> 7 THEN textflag = 0**

**menu0 = MENU (0)**

**MENU**

**WHILE menu0 = 0**

**menu0 = MENU (0)**

**dummy = MOUSE (0):GOSUB cursorposition**

**PENMODE 10**

**PENSIZ size,size**

**ON shape GOSUB free,box,roundbox,oval,  
straightline,dummy,text**

**WEND**

**ON menu0 GOSUB menu1,menu2,menu3,  
menu4,menu5,menu6,menu7**

**free:**

**MOVETO MOUSE (1), MOUSE(2)**

**PENMODE 8**

**WHILE MOUSE(0) <0 : LINETO MOUSE (1), MOUSE  
(2): WEND**

**RETURN**

**box:**

**IF MOUSE(0) > -1 THEN RETURN start**

**WHILE MOUSE (0) < 0**

**GOSUB getpoints**

**FRAMERECT VARPTR(rect(0))**

**FRAMERECT VARPTR(rect(0))**

**WEND**

**PENMODE 8**

**FRAMERECT VARPTR (rect(0))**

**RETURN**

**roundbox:**

**IF MOUSE(0) > -1 THEN RETURN start**

**WHILE MOUSE (0) < 0**

**GOSUB getpoints**

**FRAMEROUNRECT VARPTR (rect(0)),20,20**

**FRAMEROUNRECT VARPTR (rect(0)),20,20**

**WEND**

**PENMODE 8**

**FRAMEROUNRECT VARPTR (rect(0)), 20,20**

**RETURN**

**oval:**

**IF MOUSE(0) > -1 THEN RETURN start**

**WHILE MOUSE (0) < 0**

**GOSUB getpoints**

**FRAMEOVAL VARPTR(rect(0))**

**FRAMEOVAL VARPTR(rect(0))**

**WEND**

**PENMODE 8**

**FRAMEOVAL VARPTR (rect(0))**

**RETURN**

**straightline:**

**IF MOUSE (0) > -1 THEN RETURN start**

**WHILE MOUSE (0) < 0**

**hstart = MOUSE (3): vstart = MOUSE (4)**

**hend = MOUSE (5): vend = MOUSE (6)**

**MOVETO hstart,vstart**

**LINETO hend,vend**

**MOVETO hstart,vstart**

**LINETO hend,vend**

**WEND**

**PENMODE 8**

**MOVETO hstart,vstart**

**LINETO hend,vend**

**RETURN**

**text:**

**IF MOUSE (0) < 0 THEN textflag = -1**

**IF textflag = 0 THEN RETURN start**

**IF MOUSE (0) < 0 THEN leftmargin = MOUSE (3):**

**MOVETO leftmargin, MOUSE(4)**

**TEXTMODE 1**

**TEXTSIZE txtsize**

**text\$=CHR\$(0)**

**OBSCURECURSOR**

**text\$= INKEY\$**

**PRINT text\$;**

**IF text\$ = CHR\$(13) THEN PRINT PTAB (leftmargin);**

**SHOWCURSOR**

**RETURN**

**menu1:**

**menu1 = MENU (1)**

**ON menu1 GOSUB clearscreen,restorepic,  
savepic,printpic,dummy,about**

**IF menu1 = 8 THEN MENU RESET:END**

**RETURN start**

**menu2:**

**menu2 = MENU (1)**

**ON MENU2 GOSUB getit,getit,putit**

**RETURN start**

**menu3:**

**menu3 = MENU (1)**

**ON menu3 GOSUB txtsize,dummy,eraser,  
clearscreen,dummy,instructions**

**RETURN start**

**menu4:**

**oldshape = shape**

**shape = MENU (1)**

**MENU 4,oldshape,1**

**MENU 4,shape,2**

**RETURN start**

**menu5:**

**oldsize = size**

**size = MENU (1)**

**MENU 5,oldsize,1**

**MENU 5,size,2**

**RETURN start**

**menu6:**

**oldfont = font**

**font = MENU (1)-1**

**MENU 6,oldfont+1,1**

```

MENU 6,font+1,2
TEXTFONT font
RETURN start

menu7:
 check = MENU (1)
 IF check <> 1 THEN check(check) = NOT check(check):
 MENU 7,check,ABS(check (check)) +1
 IF check = 1 THEN FOR i = 2 TO 8: MENU 7,i,1: check (i) =
 0: NEXT: face = 0
 IF check <> 1 AND check (check) THEN MENU
 7,1,1:check (1) = 1:face = face + 2 ^ (check-2)
 ELSE face = face - 2^(check-2)
 IF face = 0 THEN MENU 7,1,2
 TEXTFACE face
RETURN start

crosscursor:
 SETCURSOR VARPTR (cross(0))
RETURN

textcursor:
 SETCURSOR VARPTR (textcursor(0))
RETURN

cursorposition:
 IF MOUSE (2) > 0 AND MOUSE (2) <= (bottom - top) AND
 MOUSE (1) > 0 AND MOUSE (1) <= (right-left)
 THEN IF shape = 7 THEN GOSUB textcursor
 ELSE GOSUB crosscursor
 IF MOUSE (2) < 0 OR MOUSE (1) < 0 THEN
 INITCURSOR
 IF MOUSE (1) > (right-left) OR MOUSE (2) > (bottom-top)
 THEN INITCURSOR
RETURN

waitforpress:
 WHILE MOUSE (0) = 0
 GOSUB cursorposition
 WEND
RETURN

erasecursor:
 SETCURSOR VARPTR (eraser(0))
 WHILE MOUSE (0) = 0
 IF MOUSE (2) > 0 AND MOUSE (2) <= (bottom - top)
 AND MOUSE (1) > 0 AND MOUSE (1) <= (right-left)
 THEN erasecursor
 IF MOUSE (2) < 0 OR MOUSE (1) < 0 THEN
 INITCURSOR
 IF MOUSE (1) > (right-left) OR MOUSE (2) > (bottom-
 top) THEN INITCURSOR
 WEND
RETURN

eraser:
 GOSUB erasecursor
 PENSIZE 16,16
 PENMODE 11
 MOVETO MOUSE (1), MOUSE (2)
 WHILE MOUSE(0) <0 : LINETO MOUSE (1), MOUSE
 (2): WEND

```

## RETURN

### getpoints:

```

 hstart = MOUSE (3): vstart = MOUSE (4)
 hend = MOUSE (5): vend = MOUSE (6)
 IF vstart > vend THEN SWAP vstart,vend
 IF hstart > hend THEN SWAP hstart,hend
 rect (0) = vstart:rect(2) = vend
 rect (1) = hstart:rect(3) = hend

```

### RETURN

### corners:

```

 tempshape = shape
 shape = 0
 PENSIZE 1,1
 PENMODE 10
 GOSUB waitforpress
 WHILE MOUSE (0) < 0
 GOSUB getpoints
 FRAMERECT VARPTR(rect(0))
 FRAMERECT VARPTR(rect(0))
 WEND
 PENMODE 8
 shape = tempshape
RETURN

```

### dimbox:

```

 DIM box (4+ (((hend - hstart) + 1) * 2 * INT(((vend - vstart) +
 16) / 16)))
 GET (hstart,vstart)-(hend,vend),box
RETURN

```

### getit:

```

 ERASE box
 GOSUB corners
 IF (hstart = hend) OR (vstart = vend) THEN DIM
 box(1):RETURN
 GOSUB dimbox
 IF MENU2 = 2 THEN ERASERECT VARPTR (rect(0))
RETURN

```

### putit:

```

 GOSUB corners
 IF (hstart<hend) AND (vstart<vend) THEN PUT
 (hstart,vstart) - (hend,vend), box, PSET: ELSE
 PUT (hstart,vstart), box, PSET
RETURN

```

### clearscreen: CLS : RETURN

### savepic:

```

 GOSUB getit
 PICTURE ON
 PUT (hstart,vstart),box,PSET
 PICTURE OFF
 ERASE box
 hstart = 53:vstart = 35:hend = 372:vend = 154
 GOSUB dimbox
 savedname$ = FILES$(0,"Name of Picture to Save")
 PUT (hstart,vstart),box
 IF savedname$ = "" THEN RETURN
 OPEN savedname$ FOR OUTPUT AS #1

```

```

PRINT #1,PICTURE$
CLOSE #1
NAME savedname$ AS savedname$,"MIKE"
RETURN

restorepic:
 INITCURSOR
 ERASE box
 hstart = 53:vstart = 35:hend = 416:vend = 186
 GOSUB dimbox
 getname$ = FILES$(1,"MIKE")
 PUT (hstart,vstart),box
 IF getname$ = "" THEN RETURN
 info:
 OPEN getname$ FOR INPUT AS #1
 image$ = INPUT$(LOF(1),1)
 PICTURE,image$
 CLOSE #1
RETURN

printpic:
 INITCURSOR
 ERASE box
 hstart = 0: vstart = 0: hend = right - left: vend = bottom - top
 GOSUB dimbox
 PICTURE ON
 PUT (0,0), box
 PICTURE OFF
 ERASE box
 hstart = 9:hend = 496:vstart = 1:vend = 160
 GOSUB dimbox
 OPEN "lpt1:prompt" FOR OUTPUT AS #1
 PUT (hstart,vstart),box
 ON ERROR GOTO errorhandle
 WINDOW OUTPUT #1
 PICTURE
 CLOSE #1
 caughtit: ON ERROR GOTO 0
RETURN

errorhandle:
 IF ERR = 52 THEN RESUME caughtit

txtsize:
 oldtextsize = txtsize
 INITCURSOR
 ERASE box
 hstart=100:vstart=100:hend=350:vend=200
 GOSUB dimbox
 WINDOW 2,"",(hstart + 11,vstart + 29) - (hend- 4,vend +
 14),-2
 returnhere:
 CLS
 TEXTFACE 0
 TEXTSIZE 12
 TEXTFONT 0
 PRINT"Enter size of text (min 4, max 127)"
 EDIT FIELD 1,MID$(STR$(txtsize),2), (20,40) -
 (50,55),1
 BUTTON 1,1,"OK",(120,30)-(170,75)
 dialogactive = 0
 WHILE NOT (dialogactive = 1 OR dialogactive = 6)

```

```

 dialogactive = DIALOG (0)
 dummy = MOUSE (0)
 IF MOUSE (1) < 20 OR MOUSE (1) > 50 OR MOUSE
 (2) < 40 OR MOUSE (2) > 55 THEN
 INITCURSOR
 IF MOUSE (1) > 20 AND MOUSE (1) < 50 AND
 MOUSE (2) > 40 AND MOUSE (2) < 55 THEN
 SETCURSOR VARPTR(textcursor(0))
 WEND
 txtsize = VAL(EDIT$(1))
 IF txtsize <4 OR txtsize >127 THEN GOSUB wrongsize
 WINDOW CLOSE 2
 PUT (hstart,vstart),box
 RETURN

wrongsize:
 txtsize = oldtextsize
 WINDOW 3,"",(hstart + 21,vstart + 55) - (hend- 14, vend + 3
),-2
 CLS
 PRINT"The size must be between"
 PRINT"4 AND 127 points."
 BUTTON 1,1,"OK",(165,20)-(215,45)
 WHILE DIALOG (0) < 1: WEND
 WINDOW CLOSE 3
 RETURN returnhere

init:
 DEFINT a-z
 DIM cross(33),eraser(33),textcursor(33),box(1)
 left = 3:top = 21:right = 507: bottom = 338
 WINDOW 1,"",(left,top)-(right,bottom),3
 MENU 1,0,1,"File"
 MENU 1,1,1,"New Picture"
 MENU 1,2,1,"Open Picture..."
 MENU 1,3,1,"Save Picture..."
 MENU 1,4,1,"Print Picture..."
 MENU 1,5,0,"-"
 MENU 1,6,1,"About Paint Simulator..."
 MENU 1,7,0,"-"
 MENU 1,8,1,"Quit"

 MENU 2,0,1,"Edit"
 MENU 2,1,1,"Copy"
 MENU 2,2,1,"Cut"
 MENU 2,3,1,"Paste"

 MENU 3,0,1,"Options"
 MENU 3,1,1,"Text Size..."
 MENU 3,2,0,"-"
 MENU 3,3,1,"Erase"
 MENU 3,4,1,"Clear Screen"
 MENU 3,5,0,"-"
 MENU 3,6,1,"Instructions..."

 MENU 4,0,1,"Shape"
 MENU 4,1,2,"Free form"
 MENU 4,2,1,"Rectangle"
 MENU 4,3,1,"Roundrect"
 MENU 4,4,1,"Ellipse"
 MENU 4,5,1,"Straight Line"
 MENU 4,6,0,"-"

```

#### **MENU 4,7,1,"Text"**

#### **MENU 5,0,1,"Pen Size"**

**MENU 5,1,2,"1 by 1"**

**MENU 5,2,1,"2 by 2"**

**MENU 5,3,1,"3 by 3"**

**MENU 5,4,1,"4 by 4"**

**MENU 5,5,1,"5 by 5"**

**MENU 5,6,1,"6 by 6"**

#### **MENU 6,0,1,"Font"**

**MENU 6,1,1,"Chicago"**

**MENU 6,2,2,"Default"**

**MENU 6,3,1,"New York"**

**MENU 6,4,1,"Geneva"**

**MENU 6,5,1,"Monaco"**

**MENU 6,6,1,"Venice"**

**MENU 6,7,1,"London"**

**MENU 6,8,1,"Athens"**

**MENU 6,9,1,"San Fransisco"**

**MENU 6,10,1,"Toronto"**

#### **MENU 7,0,1,"Style"**

**MENU 7,1,2,"Plain"**

**MENU 7,2,1,"Bold"**

**MENU 7,3,1,"Italic"**

**MENU 7,4,1,"Underline"**

**MENU 7,5,1,"Outline"**

**MENU 7,6,1,"Shadow"**

**MENU 7,7,1,"Condensed"**

**MENU 7,8,1,"Extended"**

#### **GOSUB about**

shape = 1

size = 1

face = 0

check = 1: check (1) = -1

font = 1

txtsize = 12

**FOR i = 0 TO 15**

    cross(i)=&H100

    cross(i+16)=&H0

**NEXT i**

cross(7)=&HFEFF

cross(32)=7

cross(33)=7

eraser(0)=&HFFFF

**FOR i = 1 TO 14**

    eraser(i) = &H8001

**NEXT i**

**FOR i = 15 TO 31**

    eraser(i)=&HFFFF

**NEXT i**

eraser(32)=0

eraser(33)=0

textcursor(0) = 0

textcursor(1) = 0

textcursor(2) = &H630

textcursor(3) = &H140

**FOR i = 4 TO 10**

    textcursor (i) = &H80

**NEXT**

textcursor(11) = &H7F0

textcursor(12) = &H80

textcursor(13) = &H140

textcursor(14) = &H630

textcursor(15) = 0

**FOR i = 16 TO 31**

    textcursor(i) = 0

**NEXT**

textcursor(32) = 11

textcursor(33) = 8

**RETURN**

instructions:

**INITCURSOR**

**ERASE box**

hstart = 0: vstart = 0: hend = right - left: vend = bottom - top

**GOSUB dimbox**

**TEXTFONT 1:TEXTSIZE 12**

**TEXTFACE 0**

**WIDTH 60**

**BUTTON 1,1,"OK",(460,290)-(500,310)**

**CLS**

**PRINT**"To draw a picture using the mouse to draw your lines or shapes, move"

**PRINT**"the cursor to where you want to start drawing and press and hold the"

**PRINT**"button down. Now move the mouse to draw on the screen. Drawing ends"

**PRINT**"when you release the mouse button. If you want to draw more than one"

**PRINT**"continuous line or shape, move the cursor to where you want and press the"

**PRINT**"button; change shape as desired from the shape menu. Line widths can be"

**PRINT**"changed from the size menu."

**PRINT:PRINT**"To copy or cut a section of the screen, choose Cut or Copy from the Edit"

**PRINT**"menu. Place the cursor at a corner of the section you want to copy or cut and"

**PRINT**"press the mouse button; while holding the button down, move the cursor."

**PRINT**"A blinking box will outline the area you have selected. When you have placed it where you want it,"

**PRINT**"release the button."

**PRINT:PRINT**"Click OK to continue."

**WHILE** **DIALOG** (0) <> 1:**WEND**

**CLS**

**PRINT**"To paste what you cut or copied, choose Paste from the Edit menu. To"

**PRINT**"paste it the same size as you cut or copied it, place the cursor where"

**PRINT**"you want the upper-left corner to be and click the mouse button. To"

**PRINT**"resize the picture, drag the rectangle as described





**DATA 0,1,15,224,7,0,128,226,0,1,15,224,7,0, 128,226**  
**DATA 0,1,15,224,7,0,128,226,0,1,15,224,7,0, 128,226**  
**DATA 0,1,15,224,7,0,128,226,0,1,15,224,7,0, 128,226**  
**DATA 0,1,15,224,7,0,128,226,0,1,15,224,7,0, 128,226**  
**DATA 0,1,15,224,7,0,128,226,0,1,15,224,7,0, 128,226**  
**DATA 0,1,15,224,7,0,128,226,0,1,15,224,4,224, 255,0**  
**DATA 224,4,224,255,0,224,7,1,0,15,226,255,0, 224,7,1**  
**DATA 0,15,226,255,0,224,7,1,0,15,226,255,0, 224,7,1**  
**DATA 0,15,226,255,0,224,7,1,0,15,226,255,0, 224,2,223**  
**DATA 0,255,13**



# *Forth Forum*

## *Forth Goes SANE*

*Jörg Langowski*  
*MacTutor Editorial Board*  
*MacTutor Vol. 1 No. 1*

One of the little unexpected features of the Macintosh that was very important to me for doing some actual calculations is the built-in floating point package. There is a full set of floating point routines, ranging from the four basic operations up to trigonometric and other functions, built right into the Mac, and they can be called up through a standard toolbox calling procedure and turn out to be reasonably (if not really breathtakingly) fast. If you want to try out the floating point routines and have access to MacForth 1.1, the following should give you some useful information. MacForth level 2 supposedly makes use of the same routines, but it was not available at the time that this was written.

Let's first look at some general features. The floating point package, called FP68K by Apple, is based on a proposed IEEE floating point standard P754. It can work on 32, 64 and 80-bit floating point numbers and 16, 32, and 64-bit integers. Some of the operations that it supports are summarized in Table 1. (Because of space limitations, this table is by no means complete, but I tried to list the most important operations.) All calculations are internally done with 80-bit accuracy; this seems spectacular, but a little overdone. Speed could probably be improved a lot, had Apple allowed for working with lower precision.

The format of a floating point number is as follows:

BYTES 1+2 : sign bit followed by  
15-bit binary exponent

BYTES 3-10 : 64-bit binary mantissa

Floating point operations are always called in the same manner: the addresses of the two arguments, source and destination, are pushed on the stack, followed by the opcode; then the toolbox trap `_FP68K` (for the basic operations) or `_ELEMS68K` (for the more complicated functions) is called. The result will be deposited in the destination address, which is overwritten. To understand the calling procedure better, let us see how toolbox traps are called from MacForth.

There are three types of toolbox traps in the Macintosh. Operating system traps expect a buffer pointer in register A0 and return an I/O result in D0; Pascal procedures expect their parameters on the stack, and Pascal functions expect a space for their result on the bottom of the stack, followed by all the parameters. MacForth 1.1 provides predefined words for all three trap types; they will be described in detail in the next issue of MacTech. The floating point package is called as a Pascal procedure, with its parameters on the stack. Since stack

items in MacForth are 32 bits in length, while some of the toolbox procedures expect 16 bit parameters, there exist three different Forth words for Pascal-type calls, depending whether there are none, one or two 16-bit items on top of the stack:

MT, W>MT and 2W>MT.

The floating point routines always expect one 16 bit parameter on top, all other parameters (up to three) are 32 bits long. The MacForth word to call these routines therefore would be W>MT. The trap addresses are \$A9EB for FP68K and \$A9EC for ELEMS68K. (These are, actually, the addresses of Package 4 and Package 5 of the Operating System Packages). We can now define our trap calls:

```
HEX
: A9EB W>MT FP68K ;
: A9EC W>MT ELEMS68K ;
```

Then we can call a floating point operation by pushing the parameters on the stack, followed by the opcode, and executing FP68K or ELEMS68K. As an example, if we had defined the 10-byte variables SOURCE and DESTINATION:

```
SOURCE CREATE 10 ALLOT
DESTINATION CREATE 10 ALLOT
```

and had put real numbers into both of them, we would add SOURCE into DESTINATION by executing:

```
SOURCE DESTINATION 0 FP68K
```

after which operation DESTINATION would contain the 80-bit result of the addition, while the contents of SOURCE would be unchanged. A function from the ELEMS68K package would be called in an equally simple way, e.g. the natural logarithm of DESTINATION is obtained by saying

```
DESTINATION 0 FP68K .
```

The 16-bit opcode consists of two parts. The high order byte gives the format of the SOURCE operand (except for opcodes \$9 and \$10, see table 1), and the low order byte specifies the operation. Listing 1 defines some Forth words that call the most important floating point operations through toolbox calls with the appropriate opcode on the stack.

There is one operation that deserves special attention, and that is the floating point - decimal conversion routine. This routine expects three parameters underneath the opcode: deepest in the stack is a pointer to a 'format record', followed by the source variable pointer, and then the destination pointer. For a



**Table 1: Summary of some floating point operations in the Mac**

**a) FP68K package**

The lower 5 bits of the opcode specify the operation:

| <u>opcode</u> | <u>mnemonic</u> | <u>operation</u>           | <u>call</u>          |
|---------------|-----------------|----------------------------|----------------------|
| \$0           | FOADD           | y:=y+x                     | X Y 0 FP68K          |
| \$2           | FOSUB           | y:=y-x                     | X Y 2 FP68K          |
| \$4           | FOMUL           | y:=y*x                     | X Y 4 FP68K          |
| \$6           | FODIV           | y:=y/x                     | X Y 6 FP68K          |
| \$8           | FOCMP           | compare                    | X Y 8 FP68K          |
| \$C           | FOREM           | y:=remainder(y/x)          | X Y C FP68K          |
| \$E           | FOZ2X           | y(80-bit):=x(other format) | X Y E FP68K          |
| \$10          | FOX2Z           | x(other format):=y(80-bit) | Y X 10 FP68K         |
| \$12          | FOSQRT          | y:=sqrt(y)                 | Y 12 FP68K           |
| \$14          | FORTI           | round y to integer         | Y 14 FP68K           |
| \$16          | FOTTI           | truncate y to integer      | Y 16 FP68K           |
| \$9           | FOD2B           | decimal string -> binary   | FORMAT DEC X 9 FP68K |
| \$B           | FOB2D           | binary -> decimal string   | FORMAT X DEC B FP68K |
| \$D           | FONEG           | negate                     | Y D FP68K            |

The upper byte of the opcode determines the format of the operand that is indicated by X in the table:

|                           |                         |                        |
|---------------------------|-------------------------|------------------------|
| \$00 - extended (80-bit); | \$08 - double (64-bit); | \$10 - single (32-bit) |
| \$20 - 16-bit integer;    | \$28 - 32-bit integer;  | \$30 - 64-bit integer  |

**b) ELEMS68K package:**

The opcodes for these functions are given in the definitions in listing 1.

conversion from a real number to a decimal number, the source variable is 10-byte floating point and the destination variable consists of a 2 byte integer, which is 0 for positive and >256 for negative numbers; a 2 byte signed integer which contains the exponent; a 1 byte field which contains the length and a 20 byte field which contains the significant digits of the mantissa. The format record is, for our purposes, a 4 byte integer which gives the number of significant digits to be used in the conversion. Example, assume that Y points to the real number 2.345, DECSTRING to a 25-byte array and FORMAT to a 4-byte integer containing 10:

**FORMAT Y DECSTRING B2D**

will leave zero in the first two bytes of DECSTRING, +9 in the next two bytes, 10 in the next byte and the ASCII characters "2345000000xxxxxxxx" thereafter (x = undefined). Vice versa, one can set up a decimal string in a 25 byte array according to these rules and call:

**FORMAT DECSTRING Y D2B**

which will leave the real number corresponding to DECSTRING in Y. The word DEC. (defined in listing 1) makes use of this conversion routine to output a real number. Try and play with those floating point routines a little; they make a very useful addition to Forth 1.1, if you want to do any kind of numerical calculations. You may want to time the floating point package using the benchmarks at the end of listing 1. I found that an 80-bit precision add takes about 0.4 ms, a multiply about 1 ms.

In the next issue of MacTech we will take a closer look at toolbox calling from Forth and at the internal structure of Forth words, so that you can write in-line assembly code for time critical operations.

## Listing 1

```
(Floating point primitives)
hex a9eb w>mt fp68k (package 4) a9ec w>mt elems68k (package 5)
(extended precision operations)
:f+ 0 fp68k ; : f- 2 fp68k ; : f* 4 fp68k ; : f/ 6 fp68k ; : x2x e fp68k ;
(double to extended operations)
:d+ 800 fp68k ; : d- 802 fp68k ; : d2x 80e fp68k ;
:d* 804 fp68k ; : d/ 806 fp68k ; : x2d 810 fp68k ;
(single to extended operations)
:s+ 1000 fp68k ; : s- 1002 fp68k ; : s2x 100e fp68k ;
:s* 1004 fp68k ; : s/ 1006 fp68k ; : x2s 1010 fp68k ;
(long integer to extended operations)
:in+ 2800 fp68k ; : in- 2802 fp68k ; : in2x 280e fp68k ;
:in* 2804 fp68k ; : in/ 2806 fp68k ; : x2in 2810 fp68k ;
(transcendental functions)
:lnx 0 elems68k ; : log2x 2 elems68k ; : ln1x 4 elems68k ; : log21x 6 elems68k ;
:expx 8 elems68k ; : exp2x a elems68k ; : exp1x c elems68k ; : exp21x e elems68k ;
:x^i 8010 elems68k ; : x^y 8012 elems68k ;
:compoundx c014 elems68k ; : annuityx c016 elems68k ;
:sinx 18 elems68k ; : cosx 1a elems68k ; : tanx 1c elems68k ; : atanx 1e elems68k ;
:randomx 20 elems68k ;
(floating point control and declarations)
:setenv 1 fp68k ; : getenv 3 fp68k ; (expect pointer to status word on stack)
:d2b 9 fp68k ; : b2d b fp68k ; (decimal <--> binary conversions, here only for 80-bit)
:single create 4 allot ; : double create 8 allot ; : float create 10 allot ;
:integer create 4 allot ; : wvar create 2 allot ; (type declarations)
wvar fstatus (floating point status word)
:fset fstatus w! fstatus setenv ; fcheck hex fstatus getenv fstatus w@ . decimal ;
(floating point output)
decimal
numstring create 25 allot ; (decimal display string)
variable zzformat (internal format for conversion routine)
numstring zzs1 (internal conversion string for dec.)
:dec. (float\format# --)
 zzformat ! zzformat swap zzs1 b2d
 zzs1 dup w@ 255 > if ." -" else ." " then
 dup 4+ count type (mantissa) 2+ w@ (get exponent)
 1 w* (convert to 32 bit integer) ." E" . ;
(floating point initialization)
0 fset (set to default extended precision)
fclear 10 erase ; : dclear 8 erase ; : sclear 4 erase ;
float z float y integer x numstring s1 y fclear z fclear 12345 x !
(benchmark programs)
variable q 1000000 x ! x y in2x x z in2x 1 x ! x y in+ z y f/ 0 q ! q z in2x y z f+
:bmark1 10000 0 do y y drop drop loop ; : bmark2 10000 0 do q @ q @ + drop loop ;
:bmark3 10000 0 do z y f+ loop ; : bmark4 10000 0 do z y f* loop ;
:bmark5 100 0 do y expx y lnx loop ; : bmark6 100 0 do y sinx loop ;
```



# Forth Forum

## A Forth Decompiler

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 2

Every one of you Mac Forth users is familiar enough with Forth to know that it is a 'threaded interpretive' language. A Forth definition (as you type it into your machine) consists of a string of other, previously defined Forth words, and is compiled as a string of addresses that point to the definitions of these other words.

This makes for a rather fast interpretation of the resulting code. However, some of the very primitive words and those words whose execution is time-critical may also be defined in machine language. MacForth has devised a very elegant way to distinguish between Forth words defined from within Forth and machine code.

### Structure of a Forth Definition

The object code of any Forth word, as it is compiled into the object dictionary, starts with at least one 16-bit word that is a meaningful, executable 68000 machine language instruction. When the Forth word is executed, the interpreter simply jumps to this address. Forth definitions (colon definitions, constants, variables) now start with one of the 68000 TRAP instructions (\$4E4X, where X can be anything from \$0 to \$F). The corresponding trap vector points to a routine which, for example, in the case of a colon definition, gets the next 16-bit word and interprets it as a Forth token (converts it to an address and executes it), or, in the case of a variable, puts the address of the variable on the stack.

If the word is defined completely in machine language, the code is executed until a special JMP instruction transfers control to the next higher level (I'll describe that later).

At this point I have to confess that I would not have come even this far if it had not been for two excellent routines that I found on a CALL-A.P.P.L.E. public domain disk. One of those - a Forth decompiler - is included below so that you can enjoy hacking into the Forth engine, the other one, disassembler, was too long to be printed here. Admittedly, part of the above sounds a bit dry and theoretical. Let's look at a simple example.

Assume you had defined the word TEST as follows:

```
: TEST DUP 2* SWAP DROP . ;
```

The Forth compiler will then create a list of 16-bit words that looks like:

|        |                             |
|--------|-----------------------------|
| \$4E4F | (trap for colon definition) |
| \$0498 | (Forth token for DUP )      |
| \$074E | ( " " " 2* )                |
| \$049C | ( " " " SWAP)               |
| \$00EC | ( " " " DROP )              |

|        |                |
|--------|----------------|
| \$0EBE | ( " " " . )    |
| \$0060 | ( " " " EXIT ) |

Interpretation always ends at the EXIT token.

### Tokens

What are those 'tokens'? They are the starting addresses of the Forth definitions that are offset by a constant that is contained in register A4 (probably to make the object code relocatable). There is a word in MacForth that converts a token to an address, **TOKEN>ADDR**. The token of a word is extracted from the vocabulary by the Forth word, **FIND**. Therefore, you will get the starting address of the example above by executing

```
FIND TEST TOKEN>ADDR .
```

The address that you'll see displayed, of course, depends on how much object code you have already in your system. Let's call this number **TESTADDR**. Then define the following word:

```
: TEST.DISP 7 0 DO I 2* TESTADDR (insert your # here) + W@ . LOOP ;
```

and execute **TEST.DISP**; you will see the list of words above. This way you can decompile any Forth word that you find in the system. The decompiler is somewhat more convenient, of course; if you use the procedure above, you still have to convert the tokens into Forth words. This is done (for one token on the stack) by executing

**NFA ID.**

This converts the *token* (not the *pfa*, as the Forth 1.1 manual says) into the name field address (NFA) and then displays the name of the word (ID.).

### Machine Language Definitions

What if the definition is direct machine language code? Again, let us look at an example, the word **SWAP**. **FIND SWAP TOKEN>ADDR** gives (in my system) \$5B60. At this address, however, we find code that does not start with a trap statement; it is a routine that does what we expect:

```
202F 0004 MOVE.L 4(A7),D0
/move item below top -> D0
2F57 0004 MOVE.L (A7),4(A7)
/move top item one down
2E80 MOVE.L D0,(A7)
```

```

/move D0 -> top of stack
4ED4 JMP (A4)
/get next token

```

We see that indeed the two top stack items are exchanged. The last statement is the end of any machine language Forth definition. This jump to the address in A4 is what I briefly mentioned above. A4 contains the address of a routine that gets and executes the next Forth token from the object code (which A3 points to):

```

MOVE (A3)+,D0
/next object token -> D0
BMI L1
/is it neg. get address from token table
JMP (A4,D0.W)
/jump to start of definition (token + A4)
L1 MOVE(A4,D0.W),D1
/get address from table
JMP (A4,D1.W)
/and jump to start of definition

```

### Hidden Definitions

When you decompile (with the program below) the word **SELECT.WINDOW**, you'll see something funny. It seems

to be a regular Forth colon definition; however, the tokens displayed seem to have no name. Only ??? and the token numbers are displayed. These are tokens whose names have been deleted from the vocabulary, but their corresponding addresses (A4+token) point to valid definitions. The reason why CSI did this is probably to keep the vocabulary short and to make words inaccessible to users whose misuse could have a disastrous effect on the system. Anyway, the word **SUBLEVEL** in the definitions below will decompile and display any such 'hidden' code, if it is a colon definition. It will display nothing for machine code definitions, you have to disassemble them.

**SELECT.WINDOW**, with this tool, then becomes very clear. Its first level definition looks like:

```

: SELECT.WINDOW {2142} {1B0E} ;

```

where the braces indicate those 'no-name' tokens. {2142} merely checks if the pointer on the stack is a valid window pointer, to keep the toolbox routine from crashing; decompile it with **SUBLEVEL** to see what it does exactly. {1B0E} is a 2-word machine code routine:

```

$A91F
/toolbox trap for SELECT.WINDOW
JMP (A4)
/and get next word. That's all!

```

## Listing 1: Forth decompiler

```

(DECOMPILER Blocks File -- Version 1.00) (ADG - modif. 110384 jl)
DECOMP (--)

```

Decompiles the definition of the next word in the input stream. A line is displayed for each word in the definition. Each line begins with its relative code address in hex. Next is the name of the word. Finally, if the word has an in-line parameter, it is shown. If the word is a branching word, the value is the target address. If the parameter is a token, its name is shown. If it is a string, the string is shown in double-quotes. If it is a word or double-word, its hex value is followed by its decimal value.

#DECOMP in block 8 can also be loaded by those who wish to write a routine to pass tokens on the stack to be decompiled. For valid tokens, its output is identical to that of DECOMP.

Written: 07/21/84 By: Alan D. Galumbeck [70220,200]

NO RIGHTS RESERVED NO RIGHTS RESERVED NO RIGHTS RESERVED

```

BASE @ DECIMAL VARIABLE HIGH.PFA 16384 MINIMUM.OBJECT 2048 MINIMUM.VOCAB : .DIGITS (n1\ n2 -
- | Types the low-order n2 digits of n1) 0 <# DO # LOOP #> TYPE ; : SPACE.TO (n -- | Spaces to column n or 2
spaces if past n) COL @ - 2 MAX SPACES ; : DISP.WORD (pfa -- pfa+2 | Display a 16-bit parameter)
DUP W@ 4 .DIGITS 31 SPACE.TO DECIMAL DUP <W@ . HEX DUP W@
NFA ?DUP IF 42 SPACE.TO ID. THEN 2+ ;
: DISP.DBL (pfa -- pfa+4 | Display a 32-bit parameter)
DUP @ DUP 8 .DIGITS 31 SPACE.TO DECIMAL . HEX 4+ ;
: DISP.STRING (pfa -- pfa+len | Display a string parameter)
34 EMIT COUNT 2DUP TYPE 34 EMIT + =CELLS ;
: DISP.TARGET (base.pfa\pfa -- base.pfa\pfa+2)
(Display a branch target and save if it's the highest)
DUP <W@ OVER + DUP HIGH.PFA @ > IF HIGH.PFA ! ELSE DROP THEN
2DUP SWAP - OVER <W@ + . 2+ ;
: DISP.TOKEN (pfa -- pfa+2 | Display a token parameter)
DUP W@ NFA ?DUP IF ID. ELSE DUP W@ 4 .DIGITS THEN 2+ ;
: DISP.ADDR (pfa -- pfa+4 | Display an address parameter)
DUP @ NFA ?DUP IF ID. ELSE DUP @ NEXT.PTR + 8 .DIGITS THEN 4+ ;

```

```

: SPECIAL.TOKENS (base.pfa\pfa\token -- [base.pfa\next.pfa] or
 [base.pfa\next.pfa0])
(Handle in-line parameters and terminating words)
CASE TOKEN.FOR EXIT OF 0 ENDOF
 TOKEN.FOR (;CODE@) OF DISP.TOKEN 0 ENDOF
 TOKEN.FOR COMPILE OF DISP.TOKEN ENDOF
 TOKEN.FOR 0BRANCH OF DISP.TARGET ENDOF
 TOKEN.FOR BRANCH OF DISP.TARGET ENDOF
 TOKEN.FOR (OF) OF DISP.TARGET ENDOF
 TOKEN.FOR (LOOP) OF DISP.TARGET ENDOF
 TOKEN.FOR (+LOOP) OF DISP.TARGET ENDOF
 TOKEN.FOR (MENU.SELECTION:) OF DISP.TARGET ENDOF
 TOKEN.FOR ALIT OF DISP.ADDR ENDOF
 TOKEN.FOR WLIT OF DISP.WORD ENDOF
 TOKEN.FOR LIT OF DISP.DBL ENDOF
 TOKEN.FOR (.) OF DISP.STRING ENDOF
 TOKEN.FOR ($LIT) OF DISP.STRING ENDOF
 TOKEN.FOR (ERROR") OF DISP.STRING ENDOF
 TOKEN.FOR (ABORT") OF DISP.STRING ENDOF
 TOKEN.FOR $ADDR OF DISP.STRING ENDOF
 (Insert the ones I've missed here.)
 0 OF 2 - DISP.TOKEN ENDOF
ENDCASE ;

: DECODE.TOKENS (pfa -- | Display the words starting at pfa)
 DUP HIGH.PFA ! DUP
 BEGIN
 HEX 2DUP SWAP - CR 4 .R 2 SPACES DUP 2+ SWAP W@ DUP NFA ?DUP
 IF ID. ELSE ." ????" drop 0 THEN
 20 SPACE.TO SPECIAL.TOKENS ?DUP
 IF FALSE ELSE DUP HIGH.PFA @ > THEN
 UNTIL
 2DROP ;

: .VALUE (n1\n2 -- | Display constants and UA variables)
 HEX.DIGITS ." hex " DECIMAL . ." decimal)" ;

: DECODE.VECTOR (pfa\vector -- | Display definition type)
 CASE
 11 OF ." User Area variable (Offset = " W@ DUP 4 .VALUE ENDOF
 12 OF ." 16 bit constant (Value = " <W@ DUP 4 .VALUE ENDOF
 13 OF ." 32 bit constant (Value = " @ DUP 8 .VALUE ENDOF
 14 OF ." Variable, array, or string" DROP ENDOF
 15 OF ." Colon definition" DECODE.TOKENS ENDOF
 ." Unknown code type (Vector = " 2 .VALUE .")"
 ENDCASE ;

: CHK.CODE.TYPE (token -- [pfa\vector\true] or [false] |
 Returns false for machine code definitions, true for others)
 TOKEN>ADDR DUP 2+ SWAP W@ DUP 16/ 1252 =
 IF 15 AND TRUE ELSE 2DROP FALSE THEN ;
(Note: 1252 is the machine code for a 68000 TRAP instruction
 divided by 16. Vector is the low-order four bits of
 the TRAP instruction.)
: sublevel chk.code.type if drop decode.tokens then ;

: DECOMP (-- | Decompile the next word in the input stream)
 GET.LINE.HEIGHT GET.TEXTSIZE BASE @ 9 TEXTSIZE 10 LINE.HEIGHT
 +FIND CR POCKET COUNT TYPE ." -- "
 IF
 IF ." IMMEDIATE " THEN
 CHK.CODE.TYPE IF DECODE.VECTOR ELSE ." Machine code definition" THEN

```

```
ELSE ." Not in dictionary" THEN
BASE ! TEXTSIZE LINE.HEIGHT CR ;

: #DECOMP (token -- | Decompile word whose token is supplied)
BASE @ GET.LINE.HEIGHT GET.TEXTSIZE 4 PICK DUP
9 TEXTSIZE 10 LINE.HEIGHT NFA ?DUP CR
IF DUP ID. ." -- " C@ 128 AND
 IF ." IMMEDIATE " THEN
 CHK.CODE.TYPE IF DECODE.VECTOR ELSE ." Machine code definition" THEN
 ELSE HEX 4 .DIGITS ." -- Not a valid token" THEN
TEXTSIZE LINE.HEIGHT BASE ! DROP CR ;
```

---



# Forth Forum

## Traps and so FORTH

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 3

First of all let me apologize for two things. The first one is an error in my article on the floating point routines; you might already have noticed it. Page 14 of MacTech 1-1, column 2, sentence 2, last word should be ELEMS68K instead of FP68K.

Secondly, I omitted something that I promised to you, namely the words for toolbox trap calling from MacForth. Let's quickly make up for this: The word that defines an operating system call is OS.TRAP. The word defined by OS.TRAP will take the address of a parameter block from the stack and put it into register A0, then execute the trap given in the definition. OS traps return their result in register D0, and the content of this register is saved in the predefined variable IO-RESULT before exiting. As an example you may define a call to the READ function as:

```
: A002 OS.TRAP READ ;
```

Calling READ with the address of a file control block on the stack will then read the appropriate number of bytes from a file or volume.

All toolbox routines that are structured like a Pascal procedure expect their parameters on the stack in the order in which they appear in the Pascal definition in the Inside Mac manual. The Forth word MT assumes that all parameters on the stack are in their correct 32-bit format, and a toolbox call defined by MT will leave the stack unchanged before executing the trap. On execution, the parameters are cleared from the stack.

If the routine expects any 16-bit parameters (16 and 32 bits are the only possibilities), it would be a little difficult to push them on the stack correctly, since MacForth generally assumes stack items to be 32 bit long (in contrast to standard FORTH-79). Most toolbox traps, however, need at most two 16-bit items, and they are at the end of the parameter list. MacForth provides the two defining words W>MT and 2W>MT for calling traps that need one or two 16-bit parameters. These parameters can be pushed on the stack as 32-bit items and are automatically adjusted to 16 bits before the trap is executed.

Last type of toolbox calls are Pascal functions. The calling convention here is that one has to leave space on the stack for the result and then push the parameters. The function will clear the parameters from the stack upon execution and leave the result in the space provided. Four types of calls are provided through the defining words FUNC>W, FUNC>L, W>FUNC>L, L>FUNC>L, for functions (in that order) that use no parameters and return a 16-bit item, no parameters/32-bit result, 16-bit parameter/32-bit result, 32-bit parameter/32-bit result.

### The Line 1111 Instruction

If you decompile and disassemble the code that is produced by the trap defining words, you will see that some of them contain instructions that start with a hex F. Here MacForth makes use of the other set of unimplemented instructions (line 1111 emulator) of the 68000!

Read this over again, it is important. The MacForth implementers have really made a fine use of the 68000 traps that are available. Not only are the 4E type traps used in making the threaded code fast (see my last article), here is the 1111 line trap, too. The Motorola 68000 Programmer's Reference Manual (you have it by now, haven't you) tells you where the trap vector is, it is hex 2C. At this location you find (in MacForth) the 32-bit number hex 5614. The code at \$5614 looks like that shown in fig. 1:

|        |                                                                                                                                                              |                                                                                                                                                                                      |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \$5614 | <pre>ADDQ.L    #2,SP MOVE.L    &lt;SP&gt;,A0 MOVE      &lt;A0&gt;+,D0 MOVE.L    A0,&lt;SP&gt; MOVE.L    0&lt;A4,D0.W&gt;,D1 JMP       0&lt;A4,D1.L&gt;</pre> | <pre>discard status register saved PC -&gt; A0 F-trap word -&gt; D0 updated PC -&gt; stack get address of trap routine rel. to A4 -&gt; D1 jump to this routine relative to A4</pre> |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**fig. 1**

Beautiful: A word that starts with hex F is a negative 16-bit number and can therefore be used as a negative offset into an address table that is below the base location that A4 points to. As an example, let's see how an OS trap defined through OS.TRAP is executed. Assume you defined READ like above. Then the word READ will generate the threaded code:

```
FFEC [F line trap for OS.TRAP]
A002 [READ OS call]
```

A4 points to \$55C4 in Forth 1.1. \$FFEC added to A4 gives \$55B0, at which location we find the address \$1850. Offset from A4, we find ourselves at \$6E14. The code there is shown in figure 2.

As you see, the trap word is moved into the routine that sets up the parameter register and executes the OS call. You may try and disassemble the definitions of the Pascal function call defining words; they also work through the 1111 line trap.

|                        |                                                                 |
|------------------------|-----------------------------------------------------------------|
| \$6E14 MOVE.L (SP)+,A1 | put PC into A1<br>(points to OS<br>trap word to be<br>executed) |
| MOVE (A1),\$1858(A4)   | get trap word &<br>put into<br>\$1858(A4) =<br>\$6E1C           |
| MOVE.L (SP)+,A0        | OS parameter<br>pointer -> A0                                   |
| \$6E1C AXXX            | Trap goes into<br>here. Yuck!<br>Self-<br>modifying code!       |
| MOVE DO,A0             | result code                                                     |
| MOVE.L A0,38(A0,D7.L)  | save in<br>IO-RESULT                                            |
| JMP (A4)               | get next Forth<br>token and<br>execute                          |

fig. 2

### 'Exotic' Procedure Calls

Though I do not recommend doing so, you might have written your own assembly language routine that requires a 16-bit parameter on the stack somewhere in between the 32 bit parameters and therefore you cannot use one of the defining words to call it. Here is a little trick that does the job. Since MacForth does not provide a word that pushes a 16-bit value on the stack and decrements the stack pointer by only two, we define our own. The base of the stack can be accessed through the variable S0, SP@ fetches the current stack pointer and SP! stores the value of S0 in the stack pointer.

```
: PUSHW S0 @ >R SP@ 2- S0 !
SP! R> S0 ! DROP ;
```

is our 16-bit push. After this, of course, the stack will be screwed up for most practical Forth purposes, so be sure to use it only if you a want to call a procedure that expects a 16-bit parameter at this position on the stack or b if you use it twice to make a composite 32-bit item on the stack from two 16-bit items.

### Windows and Controls In MacForth

The last topic that I want to deal with in this issue are some points about windowing on the Macintosh. Those of you who read David Smith's article on assembly language have probably already got a feeling how involved this can get. Even setting up a window takes quite some effort. Fortunately, the MacForth system takes most of the load off our shoulders.

The FINGER.PAINT example in the MacForth manual illustrates the ease with which basic windowing functions such as opening, closing, dragging, resizing and tracking the mouse are handled; the word DO.EVENTS takes care of many of these functions. However, the MacForth system support does not go beyond those basics. More complicated things like scrolling must be handled explicitly by you, the user. The reason for this is simply that opening, closing, dragging and sizing are functions that are handled by the Window Manager, while a scroll bar is a control for which the Control Manager is responsible. Controls are supported by MacForth Level 2; Level 1 does not document controls, although some of the definitions that are used in control handling are already present in Level 1.

For example, you can add scroll bars to any window by adding the constants SCROLL.UP/DOWN or SCROLL.LEFT/RIGHT to the window attributes. Try this in the windowing example from the Level 1 manual and then click the mouse in one of the scroll bars.

Of course, the bars don't scroll anything since DO.EVENTS does not support them. In order to be able to use these and other controls, we have to go through the Control Manager. Level 1 owners, you don't have to quit here: controls are a standard concept of the Macintosh that is supported by the Mac system and may be used from anywhere: FORTH, machine language, Pascal, C, even BASIC. It is only a little simpler in Level 2 because some of the things that we have to define for ourselves are already there.

In the next issue we'll look at the Control Manager and how to use it from Forth.





# Forth Forum

## Controls from Forth

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 4

In this month's column I want to introduce to you some concepts of control handling on the Macintosh under MacForth. Controls, as you might have guessed by now, are a very basic concept of the Macintosh user interface: a major part of mouse input to an application is done through controls.

The four main types of controls that show up to you when you work with an application are *buttons*, *check boxes*, *radio buttons* and *dials*, and these are the ones that we will be talking about this time.

All control handling routines are built into the toolbox ROM. Therefore, one can call them - in principle - from any programming environment as toolbox traps. MacForth, of course, makes this job a little easier by providing a large number of predefined routines. Interestingly enough, most of these routines are already present in Version 1.1, even though the manual says that 1.1 does not support controls. I suppose this was a typo. They wanted to say that it does not document controls. Well, the Inside Macintosh manual does that very extensively and as you will see, it is not hard to transfer those concepts into Forth.

Let's first look at the definitions of the four control types, as they are given in IM:

**Buttons** are the little rounded rectangles that have a text inside. When you click a button with the mouse, you tell the application to perform some action as shown in the example below.

"Yes" and "No" are buttons and you may choose either.

**Do you really want to erase  
the Toolbox ROM?**



**Check boxes** show the state of a parameter that can be on or off, and can be clicked on/off with the mouse:

☐ **Toolbox ROM erased**

**Radio buttons** are sets of buttons, only one of which may be on. They are used to set and display parameters that can assume discrete values or states, such as integer numbers:



**Dials**, the scroll bar being the most familiar example, are used to set and display a parameter that can change continuously (or quasi-continuously, such as the position in a text file).

Generating a control in your active window is very simple, requiring only one toolbox call. Just putting a control into a window like that, however, is useless. You can click and point at it as long as you wish, but it won't do anything unless you tell it what to do. Clicking a control only generates an event that your program has to recognize. For instance, if a radio button is clicked you have to make sure that the corresponding parameter is set on and all others that belong to the same group of radio buttons are turned off. If the page region in a scroll bar is clicked you have to tell the system what to do, change the parameter value and show the scroll bar with its new value.

But then, there is not much more to do. The Control Manager will automatically determine if you clicked a control and which one, and will also 'track' the control, that is, display it changing while you perform some action on it with the mouse. It will also, if appropriate, inactivate a control, which is then displayed in a different way and any actions on it are ignored (like the scroll bars in the Finder windows if all the icons fit into the window).

Let's assume we want to generate a window that contains the following controls:

- 1) a scroll bar that can be changed in a range from 0 to 300
- 2) a check box used to turn the scroll bar on and off.

An example program that does these functions is given below, and most of it should be self-explanatory. The Toolbox function **NewControl** that creates a new control in a window, however, has to be explained in a little more detail.

It expects 10 parameters on the stack, from bottom to top:

- 4 bytes for the function result *ControlHandle*
- *WindowPtr*, the pointer to the window that the new control belongs to
- *boundsRect*, the address of a rectangle that gives the control's size and location in the window's local coordinates
- *title*, a pointer to the control's title
- *visible*, a boolean value (16 bit); if TRUE, the control is actually drawn inside the window, otherwise, it's invisible
- *value*, a 16-bit integer, the control's initial value
- *min*, 16-bit integer, the control's minimum value
- *max*, 16-bit integer, the control's maximum value
- *procID*, 16-bit integer, a number that defines the type of control: 0 = simple button, 1 = check box, 2 = radio button, 16 = scroll bar.
- *refCon*, 32-bit integer, a number that you may use as a unique reference number: it is not used by the Control

Manager at all.

This set of parameters is rather impressive, and you can immediately see that none of the standard toolbox trap defining words in MacForth can be used to call this function. First, only function calls with at most one parameter are supported, for the others you have to include space for the result on the stack and use the MT defining word. Second, there are five 16-bit parameters between 32-bit ones, and Forth normally pushes 32-bit items on the stack.

MacForth 2.0 has the word **ADD.CONTROL** built in, which is called with the stack set up like above, only the integers are all 32-bit. **ADD.CONTROL** then converts *visible*, *value*, *min*, *max*, and *procID* to 16 bits and calls **NewControl**, which creates the control. However, if you have MacForth 1.1, you can define your own **ADD.CONTROL** by using the 16-bit push **PUSHW** that was defined in the last MacTech.

The example below contains the definitions of **PUSHW** and **ADD.CONTROL**.

The scroll bar that we define is horizontal. The up button and page-up regions of a horizontal bar are on the left side, but the bar is incremented to the right (an inconsistency in the Control Manager). Therefore the program decrements the control's value when the up button is clicked and increments it on the down button.

The only other control in the example is a check box that turns the scroll bar on and off. I have not included simple buttons or radio buttons; you may want to try them out on your own by setting *procID* to different values. Remember, if you define a set of radio buttons, your program has to keep track of which one is on and turn the others off. There is no way how the Control Manager could know which buttons belong to the same group.

---

( Demo program for using the Control Manager from Mac Forth )  
( © 1985 MacTutor by J. Langowski )

```
(pushw, new.control) hex
: pushw s0 @ >R sp@ 2- s0 ! sp! r> s0 ! drop ;
A954 mt (new.control)
: add.control
 >r (refCon) >r (procID) >r (max) >r (min) >r (value)
 pushw (visible) r> pushw (value)
 r> pushw (min) r> pushw (max) r> pushw (procID) r> (refCon)
 (new.control) ;

(test window for new.control) (JL Jan 1985)
decimal new.window my.window 50 100 200 450 my.window w.bounds
close.box size.box + my.window w.attributes my.window add.window
50 50 75 300 rect my.scroll
(set up scroll bar)
0 my.window my.scroll
" " (no title)
-1 (visible=true)
100 (value)
0 (min)
300 (max)
16 (procID for scroll bar)
here (to get a unique ref #)
add.control
constant my.control in.heap (so that handle is released when FORGETting)

(on/off box for test window)
25 50 40 200 rect button.a.rect (check box)
0 my.window button.a.rect " Scroll Bar Off" -1 0 0 1 1 (check box)
here add.control
constant my.button in.heap my.window show.controls
80 50 95 150 rect text.rect (for output)

(control tracking)
(track control, given a control handle and mouse position, tracks a control)
: my.track this.control @ @mouse track.control ;

(get.control, given a control handle, gets the value of that control)
: show.value my.control get.control text.rect erase.rect
 0 textfont 50 90 move.to ." Value:" . " " ;
```

( set.control, given a control handle and an integer, sets the control to that integer value )

( scroll bar )

```
: do.thumb my.track drop get.window show.controls show.value ;
: inc.scroll this.control @ dup get.control 1+ set.control show.value ;
: dec.scroll this.control @ dup get.control 1- set.control show.value ;
: inc10.scroll this.control @ dup get.control 10 + set.control show.value ;
: dec10.scroll this.control @ dup get.control 10 - set.control show.value ;
```

( scroll bar )

```
: do.scroll.bar case in.thumb of do.thumb endof
 up.button of dec.scroll endof down.button of inc.scroll endof
 page.up of dec10.scroll endof page.down of inc10.scroll endof
endcase ;
```

( check box )

( hilite.control, given control handle and integer value, sets the control active if value=0 and inactive if value=255 )

```
: do.button drop my.track if this.control @ dup get.control 1 swap -
 my.control over 255 * hilite.control set.control then
get.window show.controls ;
```

( control actions )

```
: do.controls swap case my.control of do.scroll.bar endof
 my.button of do.button endof
endcase ;
```

( event loop for test window )

```
: my.action show.value if begin do.events
 case mouse.down of @mouse.dn get.window
 find.control ?dup if do.controls then endof
 endcase again
else ." My Window Deactivated" then ;
```

```
my.window on.activate my.action
```



# Forth Forum

## Disk Directories & File Editing

Jörg Langowski  
MacTutorial Editorial Board  
MacTutor Vol. 1 No. 5

This month we are going to look at the organization of the 400 K bytes on the standard Macintosh disk. The operating system does a very good job of hiding this organization from you, but for patching disks, changing file attributes, and looking at files of unknown structure it is very convenient to know a little more about the 'deep structure' of the Macintosh disk.

Fortunately, it is very easy to read any byte at any position on the disk. The toolbox routines **READ** and **WRITE** do not make any distinction between files and whole disks. Let's recall how a read or write through a toolbox call is done. The toolbox traps are **A002** for read and **A003** for write. You have to set up a file control block, pass its address in register **A0**, and execute the trap. File I/O, direct disk I/O, serial I/O and even the sound generation are all handled through this mechanism. The only difference is in the file control block (FCB). It has the following structure:

|               |   |                                              |
|---------------|---|----------------------------------------------|
| Bytes 0...11  | : | Header; IM tells us nothing about it         |
| Bytes 12...15 | : | Address of the I/O completion routine        |
| Bytes 16...17 | : | I/O result code ( also returned in D0 )      |
| Bytes 18...21 | : | Pointer to filename string ( for files )     |
| Bytes 22...23 | : | Drive number ( for direct disk I/O )         |
| Bytes 24...25 | : | Reference number ( explained later )         |
| Bytes 32...35 | : | Pointer to data buffer                       |
| Bytes 36...39 | : | Requested byte count for I/O operation       |
| Bytes 40...43 | : | Actual # of bytes read/written               |
| Bytes 44...45 | : | Positioning mode, 0 : relative, 1 : absolute |
| Bytes 46...49 | : | Position offset ( in bytes )                 |

If you want to do file I/O, you have to open the file first. This is done by setting up the FCB with a valid file name (with optional volume prefix) and calling the trap **OPEN** ( **A000** ). This will return a reference number (positive 16-bit integer) in the FCB, through which all read/write calls are made from now on.

The only difference between doing a file I/O and direct device I/O operation is this reference number. The predefined Macintosh device drivers have negative reference numbers. They are listed on pages 22 and 23 of the Device Manager Programmer's Guide in IM, and the important one for us is the disk reference number, -5. If you set up the FCB like above,

with the reference number -5, and then do a **READ** or **WRITE** call, the disk will be read/written directly. That is, the operating system treats the whole disk as one large file 400K bytes long. The position from which I/O starts is given by the offset in bytes 46...49, and the number of bytes to I/O is in 36...39. If 44...45 contain a 0, the offset is counted from the last byte read/written; if it is one, it is counted from the start of the disk. After the I/O is completed, a result code will be returned in the FCB and in register **D0**. Zero means that everything went OK; a negative return code means that something was wrong. For instance, if you try to read or write to a non-existing position on the disk, -67 is returned; -50 is returned if the number of bytes actually read into the buffer is greater than the number requested. This happens if you don't read an integer multiple of 512 bytes; the number is then rounded up to the next 512 bytes.

In **FORTH** we call the traps through the defining word **OS.TRAP**. The FCB address is then passed on the stack, and the result code is stored into the variable **IO-RESULT**.

This is about all the information you need to understand the simple disk editor program that is listed at the end. It is menu-oriented and reads, writes, dumps to screen and modifies any 1024 byte block on the (internal) disk. Using that program, you can verify very easily what I am telling you in the rest of this article.

Macintosh disks are read and written by the operating system in 512 byte blocks ('logical blocks'). However, the operating system refers to 1K blocks as the smallest unit ('allocation block'). Therefore, the program reads 2 logical blocks at a time and the block number that the program asks for is  $\text{INT}(\text{logical block number} / 2)$ .

### A Directory Entry

With the described program it is now quite easy to figure out some facts about Mac disk organization; the IM manual helps, too. Starting with block 0 as the first block on the disk, the directory resides in logical blocks \$4 to \$B (allocation blocks \$2 to \$5); easily recognizable because all the file names are there. The map in Fig. 1 shows the structure of a directory entry.

The first part of the entry tells the system several parameters it needs to know about the file. 'Attributes' contains 8 bits of file attributes. For instance, Bit 7 set means that the file is open, bit 0 set means it is software locked, . Bit 6 is the copy protect bit. If you reset this one to zero, you will be able to copy a 'protected' file by dragging the icon. Bytes \$2-\$5 give the file type, such as **APPL** (application), **ZSYS** (system file) or **TEXT** (text file) in ASCII format,

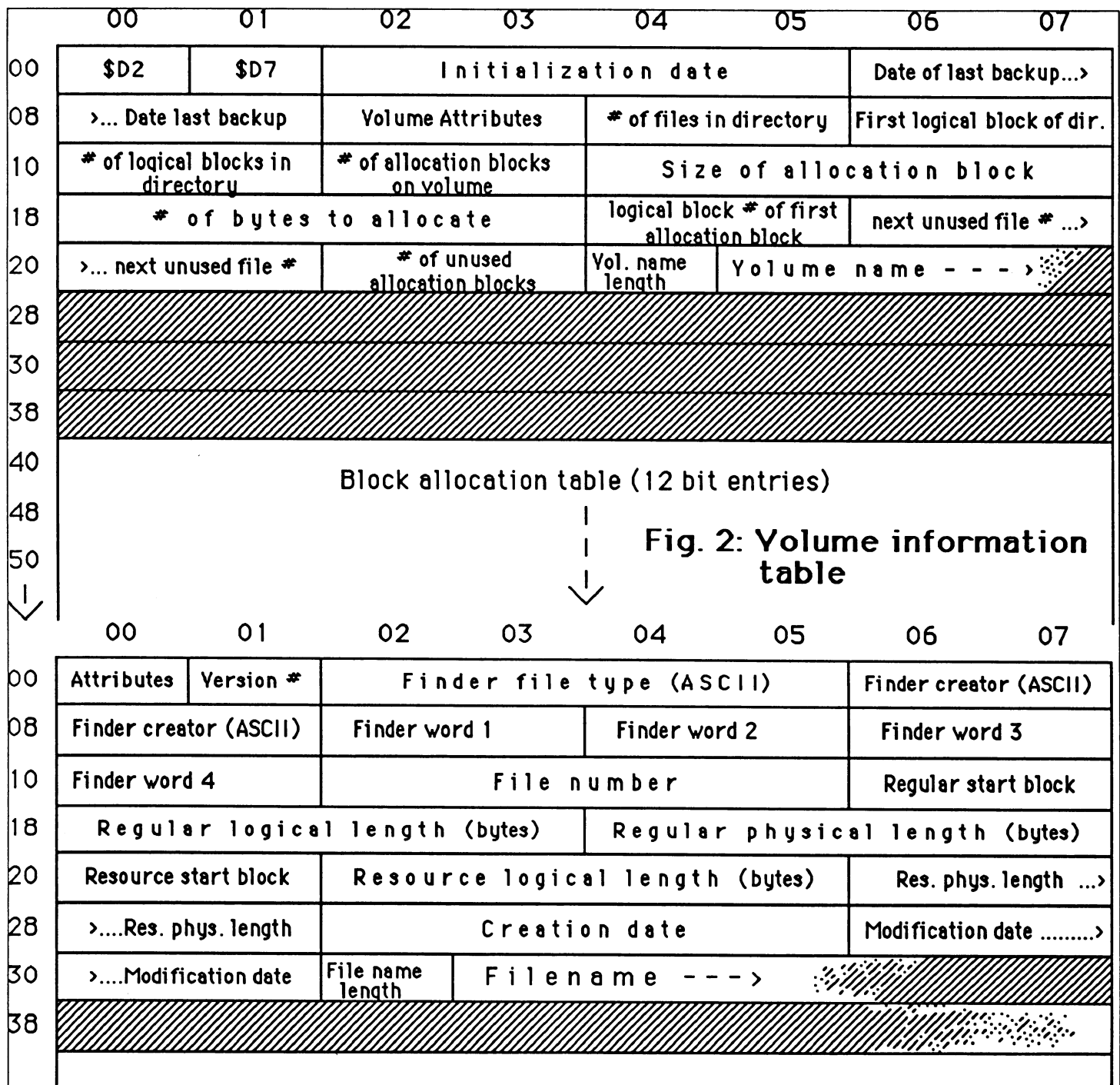


**Fig. 1: Macintosh disk directory**

bytes \$6-\$9 give the creator. The four Finder words contain information that is used by the Finder internally. All directory entries are numbered sequentially (\$14-\$15).

Bytes \$16 and \$17 (16 bit integer) give the starting block of the data fork, bytes \$18 to \$1B (32 bit integer) the length in bytes, and bytes \$1C to \$1F this length rounded up to the next 512 byte boundary. (The blocks referred to in the directory entry are allocation blocks; block number 2 starts right after the last directory block). The resource fork is

referenced in the same way by the next 10 bytes in the directory entry. The creation and modification dates of the file are kept in the next 8 bytes.

The last part of the entry gives the file name; remarkable here is that the directory entries are not all the same length. Since file names may be up to 255 characters long, reserving the maximum space for every file name would be inefficient; therefore the name is stored as a standard string starting at byte \$32 (Hex) with a length byte and the name thereafter.

## The Volume Information Table

Logical blocks \$2 and \$3 (Fig. 2) on the Macintosh disk contain information about the disk itself and a block allocation table that tells the system which blocks are in use.

The first two bytes are always \$D2D7; if they are not, the disk will not be recognized as a Macintosh disk. Following that are two 4-byte words that give the time and date of initialization and last backup. The 16-bit word Volume Attributes will have bit 7 set if the write protect latch is set on the diskette and bit 15 set if the disk is locked by software. The volume copy protection bit is also located here, it is bit 14 and if you reset it, the disk will be copyable with the Disk Copy routine on the System Disk,

regardless of whether individual files are 'protected' or not. The next entries give the total number of files in the directory, the first logical block of the file directory and the number of logical blocks in the directory.

Following are the total number of allocation blocks on the volume and the size of the allocation block in bytes (\$0400 on a standard Macintosh disk). The meaning of the remaining parameters should be clear from the diagram.

IM describes how the volume allocation block map is organized; I'll quickly repeat that here. Every allocation block (1024 bytes) is represented by a 12-bit entry. If this entry is zero, the block is unused. If it is used in a file, it contains the number of the next block in the file. The last block in the file is indicated by a 1.

```
: disk.editor ;
18 field +fcb.name 22 field +fcb.drive 24 field +fcb.vrefnum
32 field +fcb.buf 36 field +fcb.request 40 field +fcb.actual
44 field +fcb.posmode 46 field +fcb.position
12 constant dsk.menu

variable vol.fcb variable vol.fnumber variable hex.asc
create this.fcb 50 allot create vol.buffer 1024 allot
hex a002 os.trap read a003 os.trap write decimal

: open.vol this.fcb dup vol.fcb ! dup +fcb.vrefnum -5 swap w!
+ fcb.drive 1 swap w! ;

: input 0 0 >in ! query 32 word convert drop ;

: dump.fcb ." Header : " 3 0 do dup i 4* + @ ." " loop cr
." completion: " dup 12 + @ . cr ." iresult : " dup 16 + w@ . cr
." filename : " dup 18 + @ . cr ." drive : " dup 22 + w@ . cr
." refnum : " dup 24 + w@ . cr ." buffer : " dup 32 + @ . cr
." request : " dup 36 + @ . cr ." actual : " dup 40 + @ . cr
." posmode : " dup 44 + w@ . cr ." offset : " dup 46 + @ . cr ;

: setup.fcb (buffer \ block# \ fcb -- fcb)
dup +fcb.posmode 1 swap w! dup +fcb.position rot 1024 * swap !
dup +fcb.buf rot swap ! dup +fcb.request 1024 swap ! ;

: read.pb (buffer \ block# \ fcb --) setup.fcb read ;
: read.disk (block# --) vol.buffer swap vol.fcb @ read.pb ;

: write.pb (buffer \ block# \ fcb --) setup.fcb write ;
: write.disk (block# --) vol.buffer swap vol.fcb @ write.pb ;

: dump.32 (start address --)
32 0 do dup i + c@ hex.asc @ if
dup 16 < if ." 0" then . else
dup 32 < if ." " drop else emit then then loop ;

: dump.buffer (buffer address --)
9 textsize 9 line.height condensed textstyle cr
32 0 do dup i 32 * dup 16 < if ." 00" else dup 256 < if ." 0" then then
dup . + dump.32 drop cr loop ;

: read.block 12 textsize 15 line.height plain textstyle cr
." Read block #: " input dup 0< if error" Negative Block #" then
```

```

read.disk io-result @ 0= not if cr ." OS Error " io-result @ . cr abort
 else ." block read" cr then ;

: write.block 12 textsize 15 line.height plain textstyle cr
 ." Write to block #: " input dup 0< if error" Negative Block #" then
 write.disk io-result @ 0= not if cr ." OS Error " io-result @ . cr abort
 else ." block written" cr then ;

: dump.block hex vol.buffer dump.buffer decimal ;

: patch.block 12 textsize 15 line.height plain textstyle cr
 ." change byte#: " hex input decimal dup 1023 >
 if ." too large" cr abort then
 vol.buffer + ." to: " hex input decimal swap cl ;

: set.hex 1 hex.asc ! 6 -1 dsk.menu item.check 7 0 dsk.menu item.check ;

: set.ascii 0 hex.asc ! 6 0 dsk.menu item.check 7 -1 dsk.menu item.check ;

: disk.menu
 0 " DiskEdit" dsk.menu new.menu
 " Read;Write;Dump;Change;-(;Hex;Ascii" dsk.menu append.items
 draw.menu.bar dsk.menu menu.selection:
 0 hilite.menu case
 1 of read.block endof 2 of write.block endof
 3 of dump.block endof 4 of patch.block endof
 6 of set.hex endof 7 of set.ascii endof
 endcase
 events on do.events abort ;

```



# Forth Forum

## Forth Questions & Answers

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 6

**Q:** How can I write text and numbers to a window and have automatic wrap-around?

**A:** For one thing, Forth does not define the simple "." or type words so as to provide an automatic CR when the end of the window is reached. This makes sense in some applications, such as printing out tables, when the window size is not known beforehand. On the other hand, it is rather annoying if one wants to test a simple definition to see half of the printout disappear off the right edge.

You might have noticed that the built-in Forth word, WORDS, does an automatic word wrap. So it almost looks like there is some hidden word in MacForth that tests for the end of the 'screen' and prints out a CR when it is reached. Let's look at the definition of WORDS, using the decompiler from an earlier issue of this journal. (I converted the BRANCHes to IFs etc. to make the code more readable). Please refer to figure 1.

The code is pretty straightforward: The address of the CONTEXT vocabulary is put on the stack and the offset to the LATEST definition added to it. Then, after making sure it is not at the end of the list (token = 0), WORDS gets the name string of the definition and, after some manipulations, types it. The manipulations are done by two of those strange tokens that have no name associated with them, {2C7C} and {2C58}. They are defined as follows:

```
: {2c7c} col @ over wmod 1+ -
 dup {2c58} col @ if spaces else drop then ;
```

```
: {2c58} col @ +
 get.window +wbounds 6+ w@
 get.window +wline.height @ 7-
 w/ > if cr then ;
```

The first one, {2C7C} tabs to the next integer multiple of n, where n is the number on top of the stack, and wraps around if it would go beyond the limit of the window. The actual wraparound is done by {2C58}, which expects the number of characters to be output on top of the stack. It then compares the final cursor position to (right window edge)/(line height - 7), which is *approximately* the last column position in the window. If the string would print beyond this last position, a carriage return is output first. The 'approximately', by the way, is the reason why WORDS sometimes prints off the screen with non-standard text sizes. The remedy is to increase LINE.HEIGHT.

So you can very easily achieve automatic wrap when outputting text. For string output (with COUNT TYPE) you may redefine:

```
: wraptyp dup 2c58 execute type ;
```

and any string you output using this word will automatically be confined to the window boundaries (more or less; in case of trouble change LINE.HEIGHT). For number output, you either have to know beforehand how many digits your number will have, or you write your number to string conversion routine and then use WRAPTYPE.

In the near future, we will solve the text output problem (and also the Cut/Paste problem) in a more elegant way by using the TextEdit functions. Stay tuned for an article on that.

**Q:** How can I get access to the toolbox routines that are not predefined in MacForth?

**A:** I hope your question has already been answered in one of my recent columns (there is this 3 months' delay between writing the letter to us and getting the answer), but I'll repeat

```
: words
 context @ ?dup
 if @ ?dup
 if dup @ + (stack now contains address of latest definition)
 begin dup dup 1+ c@ swap c@ or while (token < 0)
 2+ count 31 and 2dup 16 {2c7c} (aha! hidden definition)
 dup {2c58} (another one) type 2 spaces
 + (add name length, get next token)
 repeat
 drop then
 then;
```

Figure 1



that important point here.

MacForth provides defining words that support most of the toolbox routines. They are the following:

**OS.TRAP** - the address on top of stack (TOS) is an operating system trap, which is compiled into the definition of the following word. On execution, this word takes the 32-bit number on TOS, puts it into register A0 and executes the trap. The result is returned in the variable **IO-RESULT**.

**MT** - the address on TOS is a Pascal *procedure* type trap. All parameters to this procedure are 32-bit items. On execution, the trap is called with all the parameters passed through the stack. They have to be set up in the order given in the trap definition (in *Inside Macintosh*).

**W>MT** - the Pascal definition expects one 16-bit item on TOS. Therefore, the (32-bit) item on TOS is converted to a 16-bit item before the trap is called.

**2W>MT** - same as **W>MT**, the two topmost items on the stack are converted to 16 bits.

The third type of traps are Pascal function type traps. This means that you have to allocate space, either 16 or 32 bits, by pushing a zero before you push the arguments on the stack. This way, function type traps can be called through **MT**. For simple functions, MacForth provides four defining words that do this space allocation automatically:

**FUNC>W** - no argument, 16 bit result (converted to 32 bits)

**FUNC>L** - no argument, 32 bit result

**W>FUNC>L** - 16 bit argument, 32 bit result

**L>FUNC>L** - 32 bit argument, 32 bit result.

For all other trap calls you have to write your custom stack set-up procedure, which is not too hard. The definition of **NEW.CONTROL** in the March issue gives an example for that. One important tool in writing such a toolbox trap definition is the routine **PUSHW** that changes TOS and the stack pointer in such a way that TOS is a 16-bit item:

```
: pushw s0 @ >r sp@ 2- s0 ! sp! r> s0 ! drop ;
```

Be cautious in using **PUSHW**; 16-bit pushes and calls that pull 16 bits have to match, otherwise the stack pointer may be 2 bytes out of frame.

**Q:** How do I use resources from MacForth?

**A:** The **BLKS** files contain no resources (as you might check using a resource editor - some of them are around now - or even a simple disk editor program). If you want to use the resource fork of a file, you have to generate the resources first with **Rmaker** (this program can also be found on several public domain disks). **Rmaker**, however, has the annoying habit of deleting the data fork of a file that is presented to him. Therefore, the correct procedure is to create a new file of the correct type and creator with **Rmaker** and put in the resources that you want. For example, the input to **Rmaker** might look like:

MyFile  
BLKSM4TH

TYPE MENU  
,1  
\14

,2  
Test  
Item1  
Item2  
(-  
Item4  
Item5

This will create the file **MyFile** with the correct type (**BLKS**) and creator (**M4TH**) and put the Apple menu (**ID=1**) and a test menu (**ID=2**) into the resource fork. Of course, this file does not contain a data fork yet, so you have to add some blocks to it by executing

```
" MyFile" 1 assign
```

```
1 open
```

```
1 10 append.blocks
```

and you can put your Forth program text into those blocks. MacForth provides the word **OPEN.RSRC** for opening the resource fork of a file; you will have to read the resources through the appropriate toolbox routines, since there are no predefined words for getting resources from a file.

Since Forth programs are usually distributed in source form, however, I do not consider resources as critical as in other applications. However, if you develop stand-alone applications in Forth (I have not used Level 3 yet, but any comments are welcome), you might want to add the resources with **Rmaker**. I will deal with using resources in one of my next columns.



Through the set of routines called TextEdit, the toolbox provides the basic functions necessary for editing and formatting text: cutting, pasting, copying, inserting, deleting and scrolling within windows.

You are already familiar with all of the functions supported by TextEdit, since almost every application makes use of them, and selecting text with the mouse is one of the basic principles of the Macintosh user interface.

In this column we shall set up a simple TextEdit (TE) record in a window under MacForth and learn about some principles of TE.

TE basically forms an interface between an ASCII text and a GrafPort, the drawing environment of QuickDraw. Given the port in which to display the text and some other parameters, which I'll soon explain, it will automatically draw the visible part of the text on the screen. TE then also responds to mouse action like clicking and dragging by moving the 'insertion point' (the blinking vertical line) and by highlighting the selected text correspondingly.

The GrafPort that we are drawing in generally is part of a window. Fig.1 reviews the structure of a window record as it is given in the Window Manager part of IM. As you can see, the GrafPort comprises the first and largest part of the window record. Most of the parameters important for drawing text and pictures are given there; however, a reference to an ASCII text record cannot be seen anywhere. This is correct; the standard convention is rather that the TE record contains a pointer to the GrafPort in which the text is to appear.

Fig. 2 shows the complete arrangement of a TE record. The text to be edited is referred to by the handle hText. TE displays this text in the *destination rectangle*; the drawing is clipped to the *view rectangle*. Both rectangles are specified in the TE record and given in the coordinates of the grafPort. The text is displayed in word wrap mode, so that words are not split when new lines are started. The start positions of new lines are kept

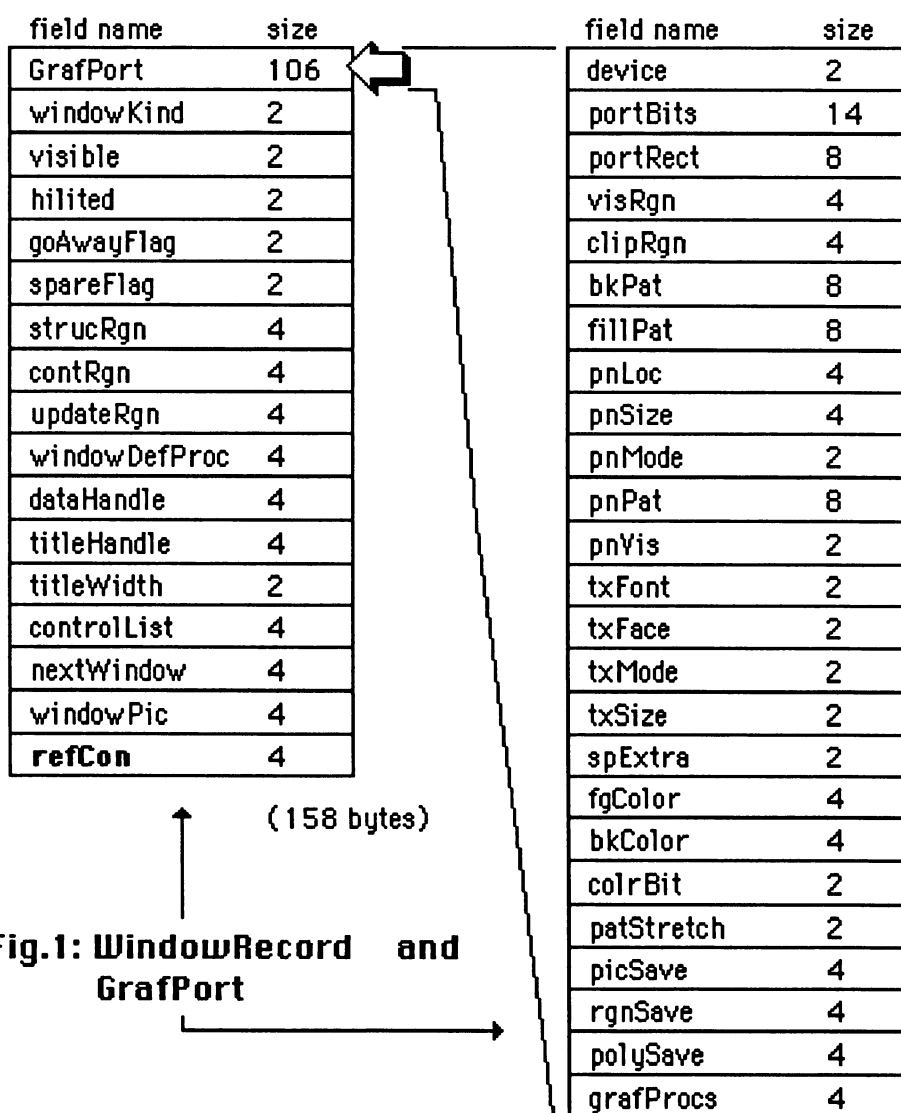


Fig.1: WindowRecord and GrafPort

in the array lineStarts and are automatically adjusted when the text is redrawn in a different destination rectangle.

The standard way to use a TE record is first to obtain a handle to a new record from the system. The toolbox trap that does this is TENew, which needs the destination and view rectangles on the stack (two 8-byte items) and returns a handle to a TE record within the current grafPort. MacForth calls this trap through the built-in compiling word TEXT.FIELD, which creates a dictionary entry that contains the TE handle as well as the current window that this particular TE record resides in. TEXT.FIELD is defined as follows:

```

: text.field (destRect\viewRect>windowPtr --)
 create get.window >r window
 tenew , get.window , r> window ;

```

Now take a look at the sample program (Listing 1). Here we define a window at the very beginning with the additional attribute **TEXT.RECORD**, which sets a flag indicating that there is a TE record in this window. The actual setup of the TE record happens close to the end of the program text, where we get the address of the content area rectangle (+WCBOUNDS) and use it both as destination and view rectangle (since we want to see everything we are typing into the window).

The new variable **TEST.TEXT** can now be used to store the TE record handle in the *refCon* field of the window record; this field is kept free for use by the application, and MacForth expects the current TE handle here.

### TextEdit Procedures

The important toolbox procedures that process TE records are **TEClick**, **TEKey**, **TECut**, **TECopy**, **TEPaste**, and some others (which are also described in IM).

Calling these procedures is somewhat simpler under MacForth than described in IM. The toolbox traps normally expect a TE handle on top of the stack. MacForth assumes that you want to edit text in the currently active window and therefore gets the TE handle that was stored in the *refCon* field. TE requests under MacForth are handled through the F line trap dispatcher (see V1.No.3; Fig.1 on p.24). The TE trap handler jumps to a self-modifying piece of code (similar to the OS trap handler) that gets the TE handle first and puts it on top of the stack, then executes the trap. Fig. 3 shows this piece of code.

```

MOVE.L (SP),A1 ;trap word -> A1
MOVE.L (A5),A0 ;QD globals ptr -> A0
MOVE.L (A0),A0 ;active port ptr -> A0
MOVE.L $98(A0),(SP) ;put TE handle-> stk
MOVE (A1),$32FE(A4) ;put trap word
XXXX <----- ;right here !
JMP (A4) ;and jump to NEXT

```

**Fig.3: Trap Handler for TE Routines**

The activate routine of the example window first calls **TEACTIVATE**, which highlights the selection range in the TE record or displays the caret (vertical line) at the text insertion point. Then the routine drops into the usual event loop. Three kinds of events are handled explicitly: mouse down, window resizing and keyboard events. Furthermore, the event loop periodically calls the routine **TEIDLE**, which blinks the caret. If a key down event is recognized, **TEKEY** is called, which inserts the character on top of the stack into the TE record at the insertion point. A mouse down event inside the editing window calls the Forth word **TEXT.CLICK**, which is an interface to the **TEClick** trap

| field name | size     | description               |
|------------|----------|---------------------------|
| destRect   | 8        | dest. rectangle           |
| viewRect   | 8        | view rectangle            |
| lineHeight | 2        | line height               |
| firstBL    | 2        | loc. of first base line   |
| selStart   | 2        | start of selection range  |
| selEnd     | 2        | end of selection range    |
| just       | 2        | justification mode        |
| length     | 2        | length of text            |
| hText      | 4        | handle to edited text     |
| txFont     | 2        | font                      |
| txFace     | 2        | style                     |
| txMode     | 2        | pen mode                  |
| txSize     | 2        | type size                 |
| inPort     | 4        | pointer to grafPort       |
| crOnly     | 2        | new line at CR only if <0 |
| nLines     | 2        | number of lines           |
| lineStarts | 2*nLines | positions of line starts  |

**Fig.2: Structure of the TE record**

and automatically handles selecting and highlighting a piece of text.

### Resizing The Window - TEUpdate

When the window is resized, we want to change the appearance of the edited text so that all of it is still visible. Therefore, we have to reset the view and destination rectangles in the TE record, using the resized window's new content area. **RECALC** in our example does this; it first moves the rectangle data from the window to the TE record and then calls **TECALTEXT**, which recalculates the line beginnings. The whole content area of the new window is then erased and marked invalid (requiring an update) so that **TEXT.UPDATE**, when called, will redraw the text correctly. (You may try and see what happens if you don't erase the window before updating or if you do not set the content area invalid).

### Scrolling Text Inside The Window

Just with the routines described so far, any text that is bigger than what the window can hold will disappear beyond the bottom of the window. Of course, we want to be able to scroll within our text so that we can see it all. I have added a vertical scroll bar to our window, and the event loop contains the necessary procedures to handle scrolling. The range of the scroll bar is fixed to 0...1000, corresponding to a display range of 1000 pixels. If you ever want to make a full-fledged

editor out of this example, you will probably want to adjust the control range to reflect the actual size of the text. Up and down buttons scroll by 10 pixels, page areas by 100 pixels each. The scrolling is done by **TESCROLL**, which scrolls the text horizontally and vertically by a relative amount, and the application itself has to keep track of the absolute position within the text. After resizing, the text starts at the beginning again, and the scroll bar is reset to zero.

### Cutting, Copying, Pasting

The menu **TE.MENU** handles cut/paste within our text record. It does not cut/paste correctly to and from other applications, since I did not add anything to handle the scrap. The MacForth handbook gives a short description of the scrap handling words, and I'll explain the scrap in a later column. Nevertheless, cutting and pasting within the example window can be done in a very simple way by calls to the **TE** routines **TECUT**, **TECOPY**, and **TEPASTE**, which do exactly what you expect from their names.

You see that the example program really forms a 'mini-editor'; if you add a routine to save/read a disk file, you have a complete small editor program.

### Vectored Execution: The DOER/MAKE Pair

Those of you who read the excellent book *Thinking FORTH* by Leo Brodie, might have noticed the **DOER/MAKE** routines for vectored execution near the end of the book. They might also have noticed that none of them work under MacForth (I know of at least one reader who has).

**DOER** is a defining word that creates a dictionary entry with a vector address in its parameter field. When a word defined by **DOER** is executed, it will place this address on the return stack, then call **EXIT** and thus jump to the routine that the vector points to.

The word that changes the vector of a **DOER** word is called **MAKE**. Suppose you define:

```
DOER TEST and
: CHANGE MAKE TEST ." HI!";
```

then **TEST** will do nothing after the first definition, and will printout "HI!" after **CHANGE** was executed. **DOER/MAKE** can therefore be used to dynamically change the execution behavior of **FORTH** code (something like a jump table in other languages).

The reason why Brodie's examples do not work with MacForth is that MacForth does not store addresses of Forth routines in the threaded code, but **tokens**, which are addresses offset from a base pointer (register **A4**).

Therefore the only thing one has to do to make **DOER/MAKE** work correctly, is to insert a **TOKEN>ADDR** into the definition of (**MAKE**) given by Brodie, so that the address and not the token of the **DOER**

word appears on the stack. Listing 2 gives the complete set of routines, modified so that they work on the Mac.

### Remarks on MacModula-2

Shortly after I had sent off my first review of Modula-2 across the ocean, I received the new official release of the MacModula-2 system by Modula Corp. Enough has been changed (and improved) that another short comment is appropriate.

The first big surprise to me was the handbook. This manual is a master example in understatement; on every other page you can read that a detailed explanation of the toolbox is 'beyond the scope of a 90 dollar product', only to find out that almost everything is explained rather concisely a couple of pages later. The existing interfaces to the toolbox routines are collected into a set of modules that can be directly used on a 512K machine, or pasted into your program on a 128K system. Of course, some module names, the module numbers for external calling and the routine numbers within the modules have changed, so our example from the last issue won't work as it is. However, it still works if you do the necessary changes.

All in all one can say that the MacModula-2 documentation almost is a mini-'Inside Macintosh'. When I work with Modula, I still have **IM** at hand, but I get most of the information about the routines (i.e. mainly how to call them) out of the Modula manual. Very convenient.

The benchmarks that I ran on the new version gave the same results as on the beta release; the speed of the interpreter does not seem to be changed. So you still can go get some coffee while a longer program is compiled, and since it is so easy to make errors in Modula, program development is still a rather lengthy process.

Rumor has it that the company that distributes MacModula-2 in Germany is working on a native code compiler for this system. Wouldn't that be nice! I only hope that they don't forget to re-compile the Compiler and Linker from M-code to native code to speed up things a little. The annual Hannover fair starts in a few weeks, and I hope they will show up there so that I can give you some more information.

### International 'Compatibility'

One of the important features of the Mac is international compatibility, or at least that's what Apple says. You are supposed to be able to modify applications to work under different languages, change keyboard configurations, to metric or non-metric, formatting on dates, etc. Well, in principle, yes... After having worked with different Macs here in Germany, I have my objections.

Of course, I have a U.S. Mac and a U.S. keyboard. I also have some disks with the U.S. System and Finder files on them. Now, if you try to start up a system with a German (or other 'international') keyboard with the US disk, it will work

fine at first. Try to type something, and you'll find that 1) not only have the keys changed which are different on that particular keyboard (such as y and z on the German board, or ; and ö, [ and ü etc.), but 2) that the whole lower row of keys is *shifted* one position. Anytime you hit a 'b', the Mac will present you with an 'n' and so on. When you hit the return key, you get the backslash, to get return, you have to press the '#' key. There are probably still more surprises, but this was already rather frustrating.

The next 'feature' (i.e. planned bug) that struck me as a little strange was that the designers of the Mac wanted to be 100% international and even changed the name of the Desktop file (which is invisible anyway) to the appropriate language. So it's 'Schreibtisch' in German, and probably (I didn't have the chance to find out) 'bureau' in French or 'scrivania' in Italian. Very cute. So what happens when you stick a U.S. disk into a system that has a German startup disk in it? Nothing, for a very long time...up to several minutes. This is because the system looks for the 'Schreibtisch' but there is only a 'Desktop'. So it assumes the 'Schreibtisch' has been destroyed and proceeds to pull all the files out of their folders and organize a new 'Schreibtisch'. So if you had some 70-odd files on your disk, you have to straighten out the mess before you can continue working. Not mentioning that the keyboard gets messed up again as soon as you start an application from your U.S. disk. So much for the touted 'international compatibility' of the Mac. Apple probably hasn't realized that people also exchange programs across country borders. Still I am very happy that I can type my ä-s, ö-s, ü-s and ß-s (and æøçfiå as well) without changing character ROMs.

### Listing 1

```
(TE Test Window)
(© 032985 MacTutor by J. Langowski)
new.window te.window
" TETest Window " te.window w.title
50 50 300 400 te.window w.bounds
sys.window te.window w.behind
close.box scroll.up/down + text.record + size.box +
te.window w.attributes te.window add.window

: text.update get.window terecord +tvirect teupdate ;
: set.inval get.window +wcbounds dup
invalid.rect erase.rect ;
: recalc get.window +wcbounds get.window terecord
over over 2 lmove 8+ 2 lmove
tecaltext set.inval text.update
get.window dup +vbar @ 0 set.control show.controls ;

(init display pointer) variable textval 0 textval !

(redisplay after scroll) : redisplay get.window dup
show.controls +vbar @ get.control dup
textval @ swap - 0 swap tescroll textval ! ;

(controls) : do.control @mouse.dn get.window
find.control ?dup if case
in.thumb of this.control @ @mouse track.control
drop (flag) redisplay endof
up.button of this.control @ dup get.control
```

```
10 - set.control redisplay endof
down.button of this.control @ dup get.control
10 + set.control redisplay endof
page.up of this.control @ dup get.control
100 - set.control redisplay endof
page.down of this.control @ dup get.control
100 + set.control redisplay endof
drop endcase then ;
```

```
(mouse handler) : do.mouse @mouse get.window
+wcbounds ptinrect if text.click else do.control then ;

: adjust.cursor @mouse get.window +wcbounds ptinrect
if ibeam set.cursor else init.cursor then ;
```

```
(main event loop) : t.edit teactivate
begin adjust.cursor teidle do.events
case mouse.down of do.mouse endof
in.size.box of wait.mouse.up drop recalc endof
?keystroke if tekey then
endcase again ;
```

```
(activate / deactivate window) : text.activate
if te.window show.window t.edit
else tedeactivate then ;
```

```
(setup edit field) : text.edit (edit field --)
dup @ swap 4+ @ swap over +wrefcon !
dup on.activate text.activate dup on.update text.update
dup terecord 0 over 72 + W! 1 swap 74 + W!
(auto new line & Geneva font) ;
```

```
(set control range) : set.scroll.range
te.window +vbar @ 0 1000 set.control.range ;
```

```
(create TE record in window)
te.window +wcbounds dup te.window text.field test.text
test.text text.edit set.scroll.range
```

```
(TextEdit menu definitions) 5 constant te.menu
0 " TextEdit" te.menu new.menu
" Cut;Copy;Paste" te.menu append.items draw.menu.bar
: te.function te.menu menu.selection: 0 hilite.menu
case 1 of tecut endof 2 of tecopy endof
3 of tepaste endof endcase ;
```

te.function

### Listing 2

```
(DOER/MAKE MacForth © MacTutor by J. Langowski)
: nothing ;
: doer create ' nothing , does> @ >r ;
variable marker
: (make) r> dup 2+ dup 2+ swap w@ token>addr 2+ !
w@ ?dup if >r then ;
: make state @ if (compiling)
compile (make) here marker ! 0 w,
else here [compile] ' !
smudge [compile] ' then ; immediate
: ;and compile exit here marker @ w! ; immediate
: undo ' nothing [compile] ' ! ;
```



# *Forth Forum*

## *Recovering Lost Files*

*Jörg Langowski*  
*MacTutor Editorial Board*  
*MacTutor Vol. 1 No. 8*

### Recovering lost Files - File Tags

Before we start the main topic of this month's column, let me make an addition to my article in the May issue (Vol1 #5). I have so far made no distinction between using MacForth 1.1 and 2.0 for any of the program examples given here. And it is true, the main difference between the two Forth versions that appears on the surface is that Forth 2.0 contains the word **ADD.CONTROL**, which you have to define for yourself in Forth 1.1 (see V1 #4).

One thing, however, does change between the two versions. The tokens of predefined words are not the same anymore. Therefore, the example that I gave in the May article (for which I happened to use Forth 2.0) does not work with Forth 1.1. In 1.1, the tokens for the two 'no-name' words are:

2C1A instead of 2C7C for the 'wrapping tab' function, and  
2BF6 instead of 2C58 for the wraparound function.

This month's example will make use of the latter function, so watch which Forth version you are using when you try it out.

Forth on the Mac can be compared to a set of very powerful, very sharp tools. You can create beautiful things with it; you can also cut yourself pretty badly. Cutting yourself with Forth can show up in a variety of disguises: the Bomb (and you may even be able to Resume!), interesting geometric patterns on the screen, requiring you to push Reset, strange machine gun type noises from the speaker, or all of the above. Of course, the MacForth editor automatically backs up screens and protects you from many disasters, but I left out something from the list... a disk drive running wild, erasing some of your last half day's hard work or even the directory.

The people who designed the Mac have provided a very clever 'safety net' for those types of crashes. The operating system not only maintains a directory of the disk which associates to each file the blocks it is written to, it keeps additional information *with each physical block*: which file it belongs to, what its position in this file is, what the attributes of that file are and when it was written. Therefore, in principle, you can reconstruct any file on the disk, even without the directory being intact, just by reading all the blocks on the disk and looking where they belong.

The information that is kept with the blocks is not much; otherwise too much disk space is lost. So you won't be able to keep track of file names this way, nor of file types or creators.

### An Update To Inside Macintosh

The extra bytes that go with each sector are called **tags**. Inside Macintosh documents that there are 12 tag bytes per block and that they contain the following data:

|       |        |                                                    |
|-------|--------|----------------------------------------------------|
| bytes | 0...3  | : file number (long word)                          |
| byte  | 4      | : file attributes (byte)                           |
| byte  | 5      | : seems to be unused                               |
| bytes | 6...7  | : sequence number of this<br>block within the file |
| bytes | 8...11 | : date block was written                           |

This table deviates a little from the one that was given in the File Manager section of Inside Macintosh (at least in the most current update that was available to me in late April, when I wrote this). Inside Macintosh puts the attributes byte at position 5 in the tags block, and says that bit 0 of byte 4 is set =1 when the file is locked, =0 otherwise. My experiments, using the example program (see below), could not confirm this. Byte 4 contains all the file attributes, including whether the fork is resource or data (bit 1 =1 or =0, resp.).

The File Tags information is not written to the file buffer that you pass to the OS routines in the file control block. Rather, after each read operation, the tags are stripped from the main part of the block and placed into a special position in memory, which is referred to by the system global TagData (=2FA). You will find the 12 tag bytes starting at TagData+2; I was not able to figure out what is contained in TagData and TagData+1. Inside Macintosh claims that the last 4 tag bytes (the block modification date) are not put into memory here, but that the logical block number appears at TagData+10 to TagData+11. This could not be reconfirmed, either; any time you read a block from disk, you will find the modification date of that block at TagData +10.

### Reading and Writing Tags

In which way do the tags influence file handling by the operating system? Not at all, as long as the directory is intact. You can not only read tags, but also by changing the memory block that contains them and writing the 512 byte buffer back to disk, modify the tags (and even give them nonsense values). This, as far as I was able to find out, does not change the behavior of the disk in any way. If you write a block on a disk back in place, with only its tags modified (saying e.g. that it belongs to a different or non-existent file), the file containing that block will still be useable.

I tried this on a copy of a MacWrite disk, changing a couple of block tags in the MacWrite as well as in the System and Finder files. The disk would still boot nicely afterwards, and also after booting with Option-Command pressed (thus reconstructing the Desktop file), nothing peculiar was seen. Therefore it seems that the block tags are really nothing but an additional safety measure to help you (or some utility routine) reconstruct a crashed disk.

This is basically what Inside Macintosh says, too. And it is also said that there would be a utility (to come) that will automatically repair a destroyed disk from the tag information. Alas, no such thing has ever crossed this editor's desk and it still seems to be one of the numerous vaporware items for the Mac. (If any of you who read this column know better, please correct me and also tell me where to get such a routine). So far, the little Forth program at the end will have to do.

### Recovering lost files

This month's example program is an update of the disk editor published in V1#5. Since I made a couple of changes to different parts of the program, it is printed in full length again. For one thing, the block length was changed from 1024 to 512 bytes, since tags refer to logical blocks, not allocation blocks. The new features that have been added are words to extract tag information from the system globals area and print it formatted; furthermore, you can dump the whole disk block by block, showing the tags of the starting block of each new file. Since files are organized on the disk in contiguous blocks, the program notices the start of a new file by the change of the file number in the tag.

The strategy for reconstructing a crashed directory, then, will be the following: First, select the drive that contains your sick disk. Then do a 'Dump all Tags' and you will get a pretty good idea of the file structure on that disk. Note that the resource fork and the data fork of a file will have the same file number but different attributes bytes; in the data fork, bit 1 equals 0, while it is set to 1 in the resource fork.

The last item in the DiskEdit menu is **RecoverFile**. This will create an output file of type TEXT (you may want to use the other drive for it, prefixing the file name by the volume name). It then asks for the starting and ending block of the file to recover, assuming that you already know (from the dump) where your file is located on the bad disk. The first block is read and it is decided whether the file to write is a regular file or a resource file. Then the other blocks are read, one by one, and written to the recovery file.

This procedure will leave you with a 'plain vanilla' document as a recovered file. You won't be able to tell the file name, whether it was an application or really a document, nor what its signature or creator is. That is too bad, but can't be helped... at least you got your file back. Good guessing helps

a lot here. You can also resort to a number of utilities to help you guess. If you happen to have a copy of ResEdit, looking at the resources of a (suspected) application will in most cases show you what application it was (from dialog boxes, strings etc.). Or, if you do not have ResEdit, you can just simply try and launch the application by double-clicking the recovered file's icon with the Command and Option keys pressed. The bomb will soon tell you whether you were wrong or right.

A copy of MacForth 1.1 that was recovered in the way explained above worked without problems, a minor annoyance being that you could start it with Command-Option-double click only. This can be changed, too: Using a utility such as SetFile (available on many public domain disks, as far as I know), you can change the Type and Creator words of your file and set the Bundle bit. For MacForth, the file type will be **APPL**, and the creator is **M4TH**. With that little trick, your file has been recovered completely; it can be launched in the usual way if it is an application, and will show the correct icon. If you don't have SetFile or a similar thing, you may as well use the disk editor published here, together with the information about directory structure given in V1#5.

### NEON: Forth-based Object-oriented Language

There is a new language coming out for the Mac: NEON by Kriya Systems from Chicago. It is too early to give a full review of this system, since at this time I have only a pre-release copy without complete documentation, but what I've seen so far looks rather impressive.

The people at Kriya have been unsatisfied with Forth for a variety of reasons, the main three points being the lack of standardized data structures like strings and arrays, lack of a 'local variable' feature (other than the stack), and bad readability due to proliferation of stack operations, i.e., lack of named formal parameters in procedure calls.

You may argue about those points; fact is, that Forth code remains rather cryptic to non-Forth programmers in many cases, and since one of the basic concepts of Forth is extensibility, it makes sense to define your own 'dialect' of Forth that does not have those (real or not) deficiencies.

The creators of NEON went one step further and built object-orientation into their dialect of Forth. That is, to each class of objects (i.e. data structures) you can define a list of methods by which to manipulate them. For instance, putting a value into an array is a completely different operation from putting it into a list, and in standard Forth you will have to define different words for both. In an object oriented language, you can simply 'tell' the object to put the value into a certain place, and the object will do whatever is appropriate. (True, **CREATE ... DOES>** has something of that flavor to it; for instance you can define arrays that do automatic range checking or that remember how many dimensions they have

etc. Still, Forth itself is a long way from being an object oriented language.)

Let me just give you a quick example of the definition of classes and objects. This is the way a class definition in NEON looks like:

```

:CLASS className <SUPER superClass
 n <INDEXED

(local variable definitions)

:M Method1: (Forth code...)
;M

:M Method2: (Forth code...)
;M

... etc. for other methods ...

:CLASS (end definition)

```

The <SUPER tells what superclass this new class belongs to (like a handle being a special kind of pointer, an unsigned integer a special kind of integer etc.); the <INDEXED tells, for indexed variables, the length of one element.

Word definitions in NEON may contain formal parameters and local variables:

```
:Word { fpar1 fpar2 \ var1 var2 var3 -- }
```

That way you can refer to stack parameters by name and have some local variables available that other routines do not know of.

Another feature that has been added is forward referencing, which also comes in handy at times. But that's all that I want to tell you about NEON so far. I do promise to write a more detailed description with more examples soon, when I have the final release with the manual.

### International Compatibility: One More Remark

Just let me add a last quick point to what I said about the keyboard problems in the last issue. Apple seems to have realized this problem; there is a program available now, Localizer, which will automatically configure your system for any of a variety of countries, so you can 'localize' a disk with e.g. a German System file on it to a US keyboard. Which is just what I needed.

### Listing 1: Disk Editor, Version 2

( Disk Editor Rev. 2, © 1985 MacTutor by  
J. Langowski )

```

: disk.editor ;
18 field +fcb.name 22 field +fcb.drive
24 field +fcb.vrefnum 32 field +fcb.buf
36 field +fcb.request 40 field +fcb.actual
44 field +fcb.posmode
46 field +fcb.position
12 constant dsk.menu 512 constant blk.size
variable vol.fcb variable vol.fnumber
variable hex.asc variable drive#
variable tag.fold 9999 tag.fold !
create this.fcb 50 allot
create vol.buffer blk.size allot
hex
 a002 os.trap read a003 os.trap write
decimal

: open.vol this.fcb dup vol.fcb !
 dup +fcb.vrefnum -5 swap w!
 +fcb.drive drive# @ swap w! ;
: input 0 0 >in ! query
 32 word convert drop ;
: input$ 0 >in ! query 32 word ;
: text.normal 12 textsize 15 line.height
 plain textstyle ;
: text.tiny 9 textsize 9 line.height
 condensed textstyle ;
hex
: need.chars 2bf6 execute ;
(Ver. 1.1; set to 2c58 for 2.0)
decimal

: dump.fcb ." Header ."
 3 0 do dup i 4* + @ ." " loop cr
 ." completion:" dup 12 + @ . cr
 ." iresult ." dup 16 + w@ . cr
 ." filename ." dup +fcb.name @ . cr
 ." drive ." dup +fcb.drive w@ . cr
 ." refnum ." dup +fcb.vrefnum w@ . cr
 ." buffer ." dup +fcb.buf @ . cr
 ." request ." dup +fcb.request @ . cr
 ." actual ." dup +fcb.actual @ . cr
 ." posmode ." dup +fcb.posmode w@ . cr
 ." offset ." dup +fcb.position @ . cr ;

hex 2FA 2+ constant tag.data
tag.data constant tag.fnumber
tag.data 4+ constant tag.attr
tag.data 5+ constant tag.lock
tag.data 6+ constant tag.sequ
tag.data 8+ constant tag.date
: s2date ?days -1 fmt.date$ type ;
: s2time ?seconds fmt.time$ type ;
 decimal
: dump.tags text.normal decimal
 ." file number : " tag.fnumber @ 6 .r cr
 ." attributes : $" tag.attr c@
 hex 2 .r decimal
 ." , sequence #: " tag.sequ w@ 4 .r cr

```



```

." date written: " tag.date @ dup s2date
 space s2time cr ;
: setup.fcb (buffer\block#\fcb -- fcb)
dup +fcb.posmode 1 swap w!
dup +fcb.position rot blk.size * swap !
 (byte pos on disk)
dup +fcb.buf rot swap !
 (buffer address)
dup +fcb.request blk.size swap !
 (# of bytes to transfer) ;
: read.pb (buffer\block#\fcb --)
 setup.fcb read ;
: read.disk (block# --)
 dup 30 need.chars ." #" . space
 vol.buffer swap vol.fcb @ read.pb ;
: write.pb (buffer\block#\fcb --)
 setup.fcb write ;
: write.disk (block# --)
 dup 30 need.chars ." #" . space
 vol.buffer swap vol.fcb @ write.pb ;
: dump.32 (start address --)
32 0 do dup i + c@ hex.asc @ if
 dup 16 < if ." 0" then . else
 dup 32 < if ." " drop else emit then
 then loop ;
: dump.buffer (buffer address --)
text.tiny cr
blk.size 32 / 0 do dup i 32 * dup 16 < if
 ." 00" else dup 256 < if ." 0" then then
 dup . + dump.32 drop cr loop ;
: read.block text.normal
 ." Read block #: " input dup 0<
 if error" Negative Block #" then
 cr read.disk io-result @ ?dup
 if cr ." OS error " . cr abort
 else ." block read" cr dump.tags then ;
: write.block text.normal
 ." Write block #: " input dup 0<
 if error" Negative Block #" then
 cr write.disk io-result @ ?dup
 if cr ." OS error " . cr abort
 else ." block written" cr then ;
: dump.block
 hex vol.buffer dump.buffer decimal ;
: patch.block text.normal
 ." change byte #: " hex input decimal
 dup blk.size 1 - >
 if ." too large" cr abort then
 vol.buffer + ." to:" hex input decimal
 swap c! ;
: dump.all.tags text.tiny 11 line.height
800 0 do i read.disk
 tag.fnumber @ tag.fold @ = not
 if cr ." new file starts, block "
 i 3 .r space decimal tag.fnumber @
 dup tag.fold ! 4 .r space hex ." $"
 tag.attr c@ 2 .r 2 spaces decimal
 then loop text.normal cr ;
: set.hex 1 hex.asc !
 6 -1 dsk.menu item.check
 7 0 dsk.menu item.check ;
: set.ascil 0 hex.asc !

```

```

6 0 dsk.menu item.check
7 -1 dsk.menu item.check ;
: drive.1 1 drive# ! open.vol
12 -1 dsk.menu item.check
13 0 dsk.menu item.check ;
: drive.2 2 drive# ! open.vol
12 0 dsk.menu item.check
13 -1 dsk.menu item.check ;
create fname$ 60 allot
: recover.file text.normal decimal cr
 ." Write output to: " input$
 dup c@ 1+ fname$ swap cmove
 fname$ 5 assign 5 create.file
 ." Recover blocks#: " input
 ." thru " input cr 1+ swap dup read.disk
 tag.attr c@ 2 and if
 ." recovering resource file, opening
 output" cr 5 open.rsrc else
 ." recovering regular file, opening
 output" cr 5 open then
 do i read.disk vol.buffer blk.size 5
 write.text ?file.error
 loop 5 close 5 remove cr
 ." File recovered, double-check before
 using." cr ;

: disk.menu
0 " DiskEdit" dsk.menu new.menu
 " Read;Write;Dump;Change;-;Hex ;
 Ascii;-;Show Tags"
 dsk.menu append.items
 " Dump All Tags;-;Drive 1;
 Drive 2;-;RecoverFile"
dsk.menu append.items draw.menu.bar
dsk.menu menu.selection: 0 hilite.menu
case 1 of read.block endof
2 of write.block endof
3 of dump.block endof
4 of patch.block endof
6 of set.hex endof
7 of set.ascii endof
9 of dump.tags endof
10 of dump.all.tags endof
12 of drive.1 endof
13 of drive.2 endof
15 of recover.file endof endcase
events on do.events abort ;
disk.menu set.hex drive.1

```



# Forth Forum

## Inline Code Speeds MacForth

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 9

### Speeding up Forth with Inline Code

When you use your computer for applications that require a lot of data shuffling and calculations, work with large arrays and matrices and so on, you tend to become a little paranoid about speed. Although Forth code is very compact through its threaded structure, and word execution (i.e. subroutine calling) is reasonably well optimized in MacForth (see MacTutor V1 No2), I have always felt uncomfortable with the overhead that goes into the execution of a simple word like **DROP**, whose 'active part' consists of one 16-bit word of machine code.

Just as a reminder: when the Forth em executes the token for **DROP** in a definition, it calls a subroutine like this:

```
DROP ADDQ.L #4,A7
 JMP (A4)
```

So it is a simple 4-byte increment of the stack pointer that does the **DROP** job. But, then the next token has to be fetched and executed by jumping to the **NEXT** routine, whose address is contained in A4, the base pointer. This makes for a several hundred percent overhead, as compared to the increment itself. This overhead is not so dramatic with other words, but it is still there: and all in all the Sieve benchmark needs 21 seconds to run in MacForth, compare this to 9 seconds in compiled C (Consulair).

How can we speed up the code? After all, we have complete control over what goes into the dictionary and could put the machine code that we need right in there, no need for time-expensive subroutine calling. This is what the Forth 2.0 assembler enables you to do. However, if you create a piece of code in Forth assembler, it tends to look much more cryptic than 'normal' assembler, which after all is readable with adequate documentation.

It would be much nicer if we had a means to create the assembly code that corresponds to a **DROP** by writing a similar word, such as **%DROP**: something like a macro. No need to worry about which registers to use, and you could use 'almost normal' Forth code for writing your routine.

It shouldn't be that difficult to persuade the Forth system to execute machine code that is embedded in a definition. Every Forth word starts with at least one executable piece of machine code, trap calls for Forth-defined words such as colon definitions and 'real' 68000 code for machine code definitions. However, this gives you either machine code or Forth, not both. Our goal is to define words that allow switching between 68000 and Forth code *within one definition*. Similar

words do exist in the Forth 2.0 assembler, but it lacks a set of macros that allow you to write inline *Forth* code instead of assembly code. Furthermore, you cannot define control structures that easily.

Assume we have Forth code that looks like this:

```
...
<token 1>
<token 2>
<token X>
<machine instruction 1>
<machine instruction 2>
...
```

etc. This sequence of instruction will get executed just fine if <token X> is a word that transfers execution to the word just following. We'll call this word **>CODE** and define it as follows:

```
:>CODE
 here 2+ make.token w, [compile] [;
 immediate
```

This word, which is *executed* during compilation, takes the next free address in the dictionary, adds 2 (this is where execution of the machine code is to start) and compiles this address as a token into the dictionary. Since a token just tells the Forth interpreter 'jump to the address that I refer to', *machine code* execution will start at the address following **>CODE**.

This is what happens at execution time. At compilation time, the words following **>CODE** in the input stream are executed, not compiled (this is what the **[COMPILE]** [ does). Therefore, if the words following **>CODE** are macros that stuff assembly code into the dictionary, you have your inline code right there.

We'll get to those macros in a minute. First, what remains is the problem how to get out of the machine code. You might recall that all machine-level Forth definitions finish with a

```
JMP (A4)
```

and the **NEXT** routine, pointed to by A4, gets the next token from the Forth code. The pointer to the next token is in register A3. Unfortunately, after we executed **>CODE**, A3 remained unchanged and still points to the word following the **>CODE** token. Which is 68000 code and certainly nothing that the interpreter will swallow. Therefore we have to reset A3 before we jump back into the Forth interpreter. This is what the word **>FORTH** does:

```
:>FORTH 47fa0004 , 4ed4 w, [compile]] ;
```

```
LEA 4(PC),A3
JMP (A4)
```

Remember, when **>FORTH** appears in the input stream, we are still in execution mode, from the preceding **>CODE** (unless we mixed things up). So **>FORTH** gets *executed* when used in a definition; it assembles code that loads A3 with the address following the JMP, then executes the JMP. Then the mode is switched back to regular Forth compilation again.

Between **>CODE** and **>FORTH** we can now place our macros that generate inline machine code corresponding to Forth primitives. The code for any of the primitives is found very easily by disassembling from the original Forth system. Of course, you may define your own code, use different registers than the MacForth definitions do or optimize the code. For instance, the built-in multiplication routine is a prime candidate for removing overhead. The routine **\***, which calls the multiplication primitive, **M\***, always does a 32- by 32-bit multiply and then *drops* the upper 32 bits of the double precision product. Some sloppiness on the part of Creative Solutions, I presume. Of course, a direct 16- by 16-bit multiply would be much faster.

I have written the macros in hex code, so that they'll work without the assembler, in case you are using Forth 1.1. The machine code is given as a comment in the program text.

### Literals

The **%LIT** and **%WLIT** macros serve as a means to put constants and addresses on the stack. They compile a long move, resp. word move instruction with the number on the stack *at compilation time* compiled as the data word(s) following the instruction. So the way to put the address of a variable on the stack in inline code is just to write: `<variable> %LIT`.

### Control Structures

The goal was to speed up the Sieve benchmark (as an example). Of course, the code would be far from optimal if we still had to use the Forth control structures; they should be coded inline, too. This means we have to keep track of addresses that we want to branch to.

The program below provides two examples, **%IF...%THEN...%ELSE** and **%DO...%LOOP**. The other control structures are not included, since they weren't necessary for this particular example. But after reading through, you should be able to write your own code for that.

**%IF** compiles a branch which is taken when the number on top of stack is zero. This branch has a zero displacement when first compiled. At the same time, the dictionary address is pushed on the *compilation time* stack (**HERE**). When

**%THEN** is encountered, the branch displacement is calculated and put into the correct address. Same holds for **%ELSE**, only that another unconditional branch is compiled that is taken at the end of the **%IF** part. This branch is resolved at the **%THEN**.

The code compiled by **%DO** takes the initial and final values from the stack and puts them on the return stack. During compilation, **HERE** is put on the stack as a reference for the backward branch taken by **%LOOP**. **%LOOP** compiles code that increments the loop counter by one and tests it against the limit; if it is still below the limit, the backward branch is taken (calculated at compilation time). **%+LOOP** behaves just like **%LOOP**, only that the increment is the number on top of stack. Note that there is one difference between **%+LOOP** and the usual Forth **+LOOP**: while the latter works with positive and negative loop increments, ours works only with positive. I did this in the interest of speed.

### The Sieve Benchmark

With all these macros available we can now recode the Sieve of Erasthones prime number benchmark into inline machine code. The changes that have to be made to the Forth code are only minor ones. At the point where the inline code is supposed to start, we insert **>CODE**; all Forth words thereafter are inline macros. They are distinguished from the regular Forth words by the preceding percent sign. When the inline part ends, we write **>FORTH** to jump back into interpreter mode.

The resulting code works (!) and executes in 9.7 seconds, as compared to 21 seconds for the Forth code.

This code is meant as an example for speeding up time-critical Forth code through the insertion of inline machine code. The words defined here are by no means a complete Forth compiler. No attempt was made to use the same words as standard Forth and do context switching; I felt that this would have been a) more complicated and b) actually confusing, because you tend to lose track of when you are in inline mode and when in interpreted Forth mode. Therefore, all inline words are compiled into the standard Forth vocabulary and have the names of the corresponding Forth words preceded by a '%'. The only control structures are **%IF...%ELSE...%THEN** and **%DO...%LOOP/%+LOOP**, where the **%+LOOP** works only for positive increments. You are encouraged to build other control structures, using the same principles.

Inline compiler definitions ( 060585 jl )  
(c) June 1985 MacTutor by J. Langowski

( inline assembly macros ) ( 060285 jl )  
hex  
: >code here 2+ make.token w, [compile] [ ;  
immediate

```
:>forth 47fa0004 , 4ed4 w, [compile]] ;
```

```

{ LEA 4(PC),A3 }
{ JMP (A4) }

: %swap 202f0004 , 2f570004 , 2e80 w , ;
{ MOVE.L 4(A7),D0 }
{ MOVE.L (A7),4(A7) }
{ MOVE.L D0,(A7) }

: %drop 588f w , ; { ADDQ.L #4,A7 }
: %dup 2f17 w , ; { MOVE.L (A7),-(A7) }
: %over 2f2f0004 , ; { MOVE.L 4(A7),-(A7) }

: %+! 205f201f , d190 w , ;
{ MOVE.L (A7)+,A0 }
{ MOVE.L (A7)+,D0 }
{ ADD.L D0,(A0) }

: %rot 202f0008 , 2f6f0004 , 0008 w ,
2f570004 , 2e80 w , ;
{ MOVE.L 8(A7),D0 }
{ MOVE.L 4(A7),8(A7) }
{ MOVE.L (A7),4(A7) }
{ MOVE.L D0,(A7) }

: %+ 201fd197 , ;
{ MOVE.L (A7)+,D0 }
{ ADD.L D0,(A7) }

: %- 201f9197 , ;
{ MOVE.L (A7),D0 }
{ SUB.L D0,(A7) }

: %i 2f16 w , ; { MOVE.L (A6),-(A7) }
: %j 2f2e0008 , ; { MOVE.L 8(A6),-(A7) }
: %k 2f2e0010 , ; { MOVE.L 16(A6),-(A7) }
{ %k is a word that does not exist in
MacForth, but is very useful to extract
a loop index one level further down }

: %+ 2017d096 , 2e80 w , ;
{ MOVE.L (A7),D0 }
{ ADD.L (A6),D0 }
{ MOVE.L D0,(A7) }

: %c@ 42802057 , 10102e80 , ;
{ CLR.L D0 }
{ MOVE.L (A7),A0 }
{ MOVE.B (A0),D0 }
{ MOVE.L D0,(A7) }

: %w@ 20574257 , 3f500002 , ;
{ MOVE.L (A7),A0 }
{ CLR.W (A7) }
{ MOVE (A0),2(A7) }

: %@ 20572e90 , ;
{ MOVE.L (A7),A0 }
{ MOVE.L (A0),(A7) }

: %c! 205f201f , 1080 w , ;
{ MOVE.L (A7)+,A0 }
{ MOVE.L (A7)+,D0 }

```

```

{ MOVE.B D0,(A0) }

: %w! 205f201f , 3080 w , ;
{ MOVE.L (A7)+,A0 }
{ MOVE.L (A7)+,D0 }
{ MOVE D0,(A0) }

: %! 205f209f , ;
{ MOVE.L (A7)+,A0 }
{ MOVE.L (A7)+,(A0) }

: %>r 2d1f w , ; { MOVE.L (A7)+,-(A6) }
: %r> 2f1e w , ; { MOVE.L (A6)+,-(A7) }

: %icl 201f2056 , 1080 w , ;
{ MOVE.L (A7)+,D0 }
{ MOVE.L (A6),A0 }
{ MOVE.B D0,(A0) }

: %lit 2f3c w , ; ;
{ MOVE.L #xxxx, -(A7) }
{ where xxxx is compiled from the stack
into the next four bytes }

: %w!it 3f3c w , w , ;
{ MOVE #xxxx, -(A7) }
{ and compile top of stack into next word }

: %< 4280bf8f , 6c025380 , 2f00 w , ;
{ CLR.L DO }
{ CPM.L (A7)+,(A7)+ }
{ BGE M1 }
{ SUBQ.L #1,D0 }
{ M1 MOVE.L D0, -(A7) }

: %> 4280bf8f , 6f025380 , 2f00 w , ;
{ CLR.L DO }
{ CPM.L (A7)+,(A7)+ }
{ BLE M1 }
{ SUBQ.L #1,D0 }
{ M1 MOVE.L D0, -(A7) }

: %= 4280bf8f , 66025380 , 2f00 w , ;
{ CLR.L DO }
{ CPM.L (A7)+,(A7)+ }
{ BNE M1 }
{ SUBQ.L #1,D0 }
{ M1 MOVE.L D0, -(A7) }

: %0= 42804a97 , 66025380 , 2e80 w , ;
{ CLR.L D0 }
{ TST.L (A7) }
{ BNE M1 }
{ SUBQ.L #1,D0 }
{ M1 MOVE.L D0, -(A7) }

: %0< 42804a97 , 6a025380 , 2e80 w , ;
{ CLR.L D0 }
{ TST.L (A7) }
{ BPL M1 }
{ SUBQ.L #1,D0 }
{ M1 MOVE.L D0, -(A7) }

```

```

: %0> 42804a97 , 6f025380 , 2e80 w , ;
{ CLR.L D0 }
{ TST.L (A7) }
{ BLE M1 }
{ SUBQ.L #1,D0 }
{ M1 MOVE.L D0,-(A7) }

: %and 201fc197 , ;
{ MOVE.L (A7)+,D0 }
{ AND.L D0,(A7) }

: %or 201f8197 , ;
{ MOVE.L (A7)+,D0 }
{ OR.L D0,(A7) }

: %if 4a9f6700 , here 0 w , ;
{ TST.L (A7)+ }
{ BEQ xxxx }
{ xxxx is a 16 bit displacement that is
 resolved by %THEN }

: %then here over - swap w! ;
: %else 6000 w, here 0 w, swap %then ;
{ BRA xxxx }
{ resolves preceding %IF and leaves new
 empty unconditional branch to be filled
 by %THEN }

: %do 2d2f0004 , 2d1f588f , here ;
{ MOVE.L 4(A7),-(A6) }
{ MOVE.L (A7)+,-(A6) }
{ ADDQ.L #4,A7 }
{ leaves HERE on the stack for back branch
 by %LOOP or %+LOOP }

: %loop 5296204e , b1886e00 ,
 here - w, ddfc w, 8 , ;
{ ADDQ.L #1,(A6) }
{ MOVE.L A6,A0 }
{ CMPM.L (A0)+,(A0)+ }
{ BGT xxxx }
{ ADDA.L #8,A6 }
{ the last instruction cleans up the return
 stack. Branch resolved in this word }

: %+loop 201fd196 , 204e w, b1886e00 ,
 here - w, ddfc w, 8 , ;
{ MOVE.L (A7)+,D0 }
{ ADD.L D0,(A6) }

decimal

(Eratosthenes Sieve Benchmark,
 inline code) (060285 jl)
8192 constant size
create flags size allot
: primes flags size 01 fill
>code 0 %lit size %lit 0 %lit
%do flags %lit %i+ %c@
%if 3 %lit %i+ %i+ %dup %i+
 size %lit %<
%if size %lit flags %lit %+
%over %i+ flags %lit %+
%do 0 %lit %ic! %dup %+loop
%then %drop 1 %lit %+
%then
%loop >forth . ." primes " ;
: 10times
1 sysbeep 10 0 do primes cr loop
1 sysbeep ;

(Eratosthenes Sieve Benchmark,
 standard version)
8192 constant size
create flags size allot
: primes flags size 01 fill
0 size 0
do flags i+ c@
if 3 i+ i+ dup i+ size <
if size flags + over i+ flags +
do 0 ic! dup +loop
then drop 1+
then
loop . ." primes " ;
: 10times
1 sysbeep 10 0 do primes cr loop
1 sysbeep ;

```



# Forth Forum

## Converting Forth Blocks to Ascii Text

### Accuracy in Modula-2 Reviewed

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No.10

With the advent of other Forth systems than MacForth, and Forth-related packages such as NEON, a utility is needed that lets us quickly convert the MacForth 'Blocks' format into a standard ASCII text file and back. NEON, for instance, expects its input from a normal text file. Also this Journal expects *its* input from a MacWrite file, and after I have written an example program, cutting it out of the Forth screens and pasting it into my column is quite cumbersome.

So here is a routine that does this job automatically (it contains some ideas out of a program on one Call-A.P.P.L.E. public domain disk). At the same time we are going to learn something about the Standard File Interface package, which is undocumented (but fully usable) in MacForth 1.1.

What do we need to know to convert files from Blocks to Text format and back? The type of a Blocks file is BLKS and the creator M4TH, a Text file such as saved by 'text only' MacWrite or MDS Edit is of type TEXT and creator MACA (this probably stands for Mac Ascii). In principle, both files contain ASCII text, which, however, is organized differently.

#### The Format of a Blocks File

MacForth blocks files consist of *screens*, each of which contain 16 lines of 64 characters. One line is always 64 characters long and padded with blanks after the last character. This format is a relic from early FORTH times and - in my opinion - rather obsolete. But since the MacForth editor uses it, we have to live with it. The length of a file is of course very simply related to the number of screens; each screen occupies exactly 1024 bytes.

Text files, on the other hand, contain lines of ASCII characters which are separated by carriage returns (CR).

One very simple thing that one can try out to communicate between the two types of files is to change type and creator of a Blocks file to TEXT and MACA, hoping that it is recognized by Edit, for example. When I tried this, it would show a file that had only one line (of course, since there are no CRs). So we do have to read the Blocks file, line by line, and create a new text file to which we write those lines.

#### Reading Forth Blocks

A Forth block in a file is accessed through the word **BLOCK**, which accepts the block number on top of stack and returns the address of a buffer that contains the block. The block is automatically read in if it is not already there.

The position of a line in the block is at  $(64 * \text{line number})$ . The word **>LINE.START** in the program example returns the address of a line in a given block. We then remove trailing blanks from this line, add a CR at the end and write it

to the Text file (which we have opened before). This is what the word **COPY.BLOCK** does, which is called from the main menu: It traverses the Blocks file line by line and writes the Text file.

#### ASCII to Blocks Conversion

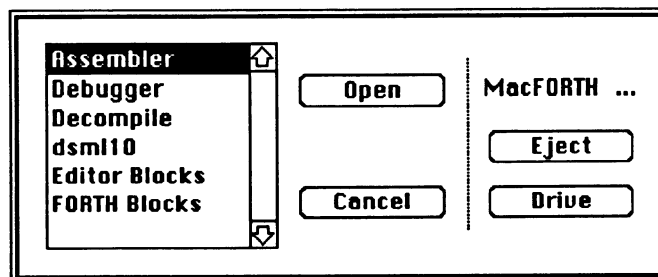
The reverse direction is a little bit more complicated. The total number of lines in a blocks file is known exactly from its size, so we know ahead of time how many screens we have to convert; also a Text file can be extended arbitrarily by just adding lines. Not so the other way around: a Blocks file must be extended by appending blocks before writing to them, and the end of a Text file is known only when an end-of-file occurs.

In the example, each text line is read into the string variable **TEXT.BUF** with the character count in the first byte and the actual line in the following bytes. This string is then copied into a new line in the buffer; when 11 lines have been processed in this way, the buffer is written out to disk (**UPDATE FLUSH**) and the Blocks file extended by one block. The last five lines of each block stay blank for easier editing. Some editing is necessary because the block boundaries will often split your definitions.

#### Choosing Input and Output Files: The Standard File Package

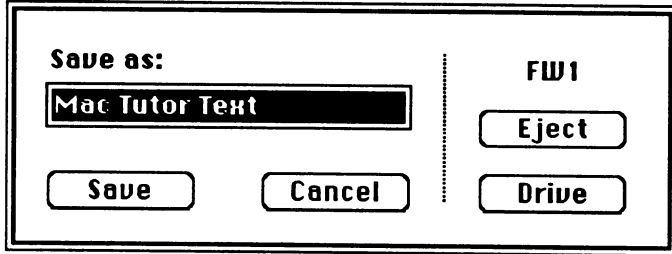
So the principle of this utility routine has been set up. But we still have to open the files that we want to work on. To make file access more comfortable, let's use the standard Macintosh file package to select those files.

You've seen the dialog boxes that make up the interface to this package many times. Whenever you are supposed to select a file for input, a Mac application that conforms to the User Interface standards will show you a list of files to select

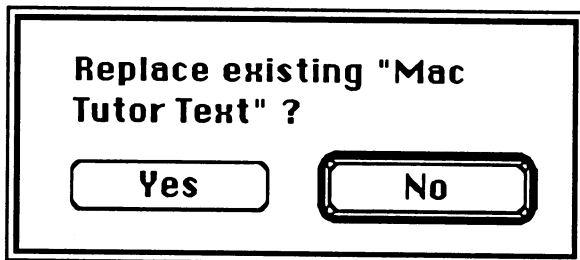


from and option buttons that let you change drives, eject disks or quit the operation.

When you select a file for output (or save) you are presented with an input box with a default file name:



and when the file name that you give already exists on that disk, the standard file package will ask for reconfirmation to overwrite that file.



All these functions are built into two toolbox routines, `SFGetFile` and `SFPutFile`. MacForth provides built-in interfaces to those routines, the words (`GET.FILE`) and (`PUT.FILE`). The Forth words expect less parameters than the corresponding toolbox traps.

(`GET.FILE`) needs, from bottom to top of stack:

- a point (2\*16 bit) where the top left of the dialog box is to be located;
- the address of a prompt string (zero for no prompt);
- the address of a list of file types to be displayed;
- the number of different file types;
- the address of a *reply record*.

(`PUT.FILE`) needs:

- the top left point;
- the address of a prompt string;
- the address of a string which may contain the default file name and later contains the actual file name;
- the address of a reply record.

Information about the user's response to the dialog box is contained in the reply record. The format is:

- 2 bytes boolean, true if OK, false if Cancel button was clicked;
- 4 bytes of file type information;
- 2 bytes integer, contain the volume reference number (important if volume was changed);
- 2 bytes integer, file version number, not used here;
- 64 bytes that contain the file name in Mac string

format (count byte plus characters).

The list of file types is an array of 4 byte integers; each 4-byte cell contains one valid type designator, such as `PICT`, `TEXT`, `DATA`, etc. On calling (`GET.FILE`), only those files corresponding to one of the designators will be displayed. Invisible files will be displayed, too.

In our program, input and output files are set up using the standard file interface by the words `TEXT.OPEN`, `BLOCK.OPEN`, `TEXT.CREATE` and `BLOCK.CREATE`. After the dialog has been processed, the volume reference number (which might have been changed from the default volume) is moved into the file's FCB and the type and creator fields are set depending on whether a Text or a Blocks file is written. Thereafter, the actual file conversion takes place.

During the execution of the conversion routines the *activate* event is switched off. If you don't do this, the main Forth window would be activated again after the first file dialog has been processed, executing an `ABORT` and jumping back to the 'ok' prompt, quitting your program. Try this out and see what happens when you leave out line 6 in the `CHANGE.MENU` definition.

With that little routine, creating MacWrite listings from your Forth programs will be easy now.

## Floating Point Glitches in Modula-2

Even though I am responsible for Forth, I must make one more Modula comment.

*Watch out when you're using math!*

I have become extremely careful in this respect. The 32-bit routines are fast, true. In order to be useful, they should also give correct results (within rounding errors). This is not so. Try the example in listing 2 (`WindowIO` is a module of my own that just sets up a standard window for I/O). The program just keeps adding smaller and smaller numbers to 1.0, which I did originally as an accuracy test. The output is shown there, too. Sometimes the output routine seems to fail, sometimes the addition itself. And I thought this was a system I could use for some of my calculations in the lab... oh well. No more comments.

Surprisingly enough, though, a non-linear curve fitting program that I wrote in Modula seems to work well, without major problems. It may be that the bug shows up only when the numbers are close to integers (???). Any comments from readers with similar experiences are appreciated, any help even more. (Richard Ohran, are you reading this? If not, a bug report will follow.)

### **Listing 1:** Converting text to blocks files

( Forth <-> Ascii Conversion) ( © 1985 MacTutor by J. Langowski )

**: tx-blks ;**

```
variable ifile# -1 ifile# !
variable ofile# -1 ofile# !
: ifile ifile# @ ; : ofile ofile# @ ;
```

```
hex 4D414341 constant "maca decimal
create ="blks "blks ,
create ="textdata "text , "data ,
```

```
18 field +fcb.name 22 field +fcb.vrefnum
32 field +fcb.type 36 field +fcb.creator
```

```
(standard file reply record fields)
00 field +good : @good +good w@ ;
02 field +ftype : @ftype +ftype @ ;
06 field +vrefnum : @vrefnum +vrefnum <w@ ;
10 field +fname
```

```
create ireply 80 allot
ireply +fname constant iname
create oreply 80 allot
oreply +fname constant oname
variable screen# variable cur.line#
create text.buf 100 allot
99 constant text.read.limit
5 constant ch.menu
```

```
: moverefnum (file# \reply --)
@vrefnum swap >fcb +fcb.vrefnum w! ;
```

```
: get.input.name
>r >r >r
100 100 xy>point 0 r> r> r@
(get.file) page r> @good
0= if abort then ;
```

```
: get.save.name
page >r >r >r
100 100 xy>point r> r> r@
(put.file) r> @good
0= if abort then ;
```

```
: text.open
next.fcb ifile# !
page ." Text file to convert:"
="textdata 2 ireply get.input.name
iname ifile assign
ifile ireply moverefnum
ifile open ?file.error
ifile rewind ?file.error ;
```

```
: block.open
next.fcb ifile# !
page ." Blocks file to convert:"
="blks 1 ireply get.input.name
iname ifile assign
```

```
ifile ireply moverefnum
ifile open ?file.error ifile select ;
```

```
: block.create
next.fcb ofile# !
" Mac Tutor Blocks"
oname over c@ 1+ cmove
" Save as:" oname oreply get.save.name
oname ofile assign
ofile oreply moverefnum ofile delete
ofile create.blocks.file ?file.error
ofile open ?file.error ofile select
2 ofile append.blocks
1 screen# ! 0 cur.line# !
0 block b/buf bl fill update flush ;
```

```
: text.create
next.fcb ofile# !
" Mac Tutor Text" oname over c@ 1+ cmove
" Save as:" oname oreply get.save.name
oname ofile assign
ofile oreply moverefnum ofile delete
ofile create.file ?file.error
ofile open ?file.error ofile rewind
ofile get.file.info
"text ofile >fcb +fcb.type !
"maca ofile >fcb +fcb.creator !
ofile set.file.info ;
```

```
: >line.start (block \line -- addr of line)
64 * swap block + ;
```

```
: write.line
>line.start 64 -trailing
over over + 13 swap c! (add CR)
1+ ofile write.text ;
```

```
: write.screen
dup 16 0 do dup i (screen \line)
over over . 2 spaces . cr (debug)
write.line loop drop (n) ;
```

```
: copy.block ifile get.eof b/buf /
0 do i . cr i write.screen
do.events drop loop ;
```

```
: read.line ifile current.position
text.buf 1+ text.read.limit
ifile read.text
ifile current.position swap -
text.buf c! ;
```

```
: copy.line
read.line ?eof not
if cur.line# @ dup .
64 * screen# @ dup . cr
block + text.buf 1+ swap
text.buf c@ 1- cmove
1 cur.line# +! cur.line# @ 10 >
if 0 cur.line# ! update flush
1 screen# +! 1 ofile append.blocks
then do.events drop true
```



```

else update flush false then ;

: copy.text begin copy.line not until ;

: >text block.open text.create copy.block ;

: >block text.open block.create copy.text ;

: change.menu
 0 " Change" ch.menu new.menu
 " Forth->Ascii;Ascii->Forth;"
 ch.menu append.items draw.menu.bar
 ch.menu menu.selection:
 1 activate.event scale -1 xor events !
 0 hilite.menu
 case 1 of >text endof
 2 OF >block endof endcase
 events on do.events abort ;

```

change.menu

### Listing 2: Modula 2 FP 'accuracy test'

```

MODULE Accuracy;

FROM InOut IMPORT ReadCard, WriteCard,
 WriteString, WriteLn, ClearScreen;
FROM RealInOut IMPORT ReadReal, GWriteReal;
FROM WindowIO IMPORT SetupWindow;
FROM EventManager IMPORT Button;
VAR a,b : REAL;

BEGIN
 SetupWindow
 ("Accuracy test",40,40,350,450,4,9);
 a := 1.0; b := 1.0;
 WriteString
 (" a b a+b"); WriteLn;

 REPEAT
 b := b/10.0;
 GWriteReal(a,12); GWriteReal(b,12);
 GWriteReal(a+b,12); WriteLn;
 WHILE Button() DO END;
 UNTIL (b = 0.0);

```

REPEAT UNTIL Button();

END Accuracy.

**Listing 3:** Output of the Accuracy program (terminated before underflow occurred). Note that b is sometimes printed as zero, although a+b seems to be correct; and for some smaller values of b (1.0E-19 to 1.0E-25) a+b starts to behave very strangely.

| a        | b          | a+b          |
|----------|------------|--------------|
| 1.000000 | 0.1000000  | 1.100000     |
| 1.000000 | 0.01000000 | 1.010000     |
| 1.000000 | 0.00000000 | 1.001000     |
| 1.000000 | 0.00010000 | 1.000100     |
| 1.000000 | 0.00001000 | 1.000010     |
| 1.000000 | 0.00000100 | 1.000001     |
| 1.000000 | 0.00000010 | 1.000000     |
| 1.000000 | 1.0000E-08 | 1.000000     |
| 1.000000 | 1.0000E-09 | 1.000000     |
| 1.000000 | 1.0000E-10 | 1.000000     |
| 1.000000 | 1.0000E-11 | 1.000000     |
| 1.000000 | 0.0000E-13 | 1.000000 ?   |
| 1.000000 | 1.0000E-13 | 1.000000     |
| 1.000000 | 1.0000E-14 | 1.000000     |
| 1.000000 | 1.0000E-15 | 1.000000     |
| 1.000000 | 1.0000E-16 | 1.000000     |
| 1.000000 | 1.0000E-17 | 1.000000     |
| 1.000000 | 0.0000E-19 | 1.000000 ?   |
| 1.000000 | 1.0000E-19 | 2.844674 ??? |
| 1.000000 | 1.0000E-20 | 1.184467 ??  |
| 1.000000 | 1.0000E-21 | 1.018447 ??  |
| 1.000000 | 0.0000E-23 | 1.001845 ??  |
| 1.000000 | 1.0000E-23 | 1.000184 ?   |
| 1.000000 | 1.0000E-24 | 1.000018 ?   |
| 1.000000 | 1.0000E-25 | 1.000002 ?   |
| 1.000000 | 0.0000E-27 | 1.000000 ?   |
| 1.000000 | 1.0000E-27 | 1.000000     |
| 1.000000 | 1.0000E-28 | 1.000000     |
| 1.000000 | 1.0000E-29 | 1.000000     |
| 1.000000 | 1.0000E-30 | 1.000000     |
| 1.000000 | 1.0000E-31 | 1.000000     |
| 1.000000 | 1.0000E-32 | 1.000000     |
| 1.000000 | 1.0000E-33 | 1.000000     |
| 1.000000 | 1.0000E-34 | 1.000000     |
| 1.000000 | 1.0000E-35 | 1.000000     |



# Forth Forum

## Solving Systems of Linear Equations

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 11

This is the first of a series of columns that will deal with the general problem of doing numerical calculations in MacForth. Forth's philosophy is to use integer arithmetic in many cases that would be handled with floating point in other languages. The reason for this has to be seen historically in the development of Forth, which first was used almost exclusively as a language to do process control. It was desirable not to have the ballast of a floating point package in implementations that used 8-bit processors with a limited amount of memory, and there is, of course, a great speed advantage in using integer arithmetic.

When used in 'custom-designed' routines for one particular problem, integer arithmetic *can* do as well as floating point. However, one has to scale all the numbers involved so that they fit into the range that is given by the 4 bytes of the Mac's integer arithmetic (or the 2 bytes of some other system). On the other hand, numbers shouldn't get too small, either, because accuracy is lost very quickly. The constant need to haul scaling factors around between parts of the program then makes the code rather hard to read and bug-prone.

Again this is the old tradeoff between speed and low memory requirement on one side and flexibility and readability on the other. If we want to write a set of mathematical routines in Forth that will be useful no matter what the particular problem is (whether distances are in nanometers or lightyears, weights in tons or micrograms) the easiest way to do this is to use floating point arithmetic. This is especially true on the Macintosh, since we have an excellent floating point package with 80-bit accuracy built in.

This FP package, also called SANE (Standard Apple Numeric Environment) conforms to the proposed IEEE standard on floating point arithmetic (see my article in MacTutor V1#1). MacForth 2.0 offers Forth code for a very slick interface to the SANE routines, using its own floating point stack and even modifying the interpreter so that real numbers are accepted as input. There are two problems with this code, though: First, we cannot print it here for obvious reasons and therefore our program would run only under MacForth 2.0, which would be a little too restricted for this Forum. Second, according to my own tests the floating point interface adds so much overhead that actual calculations are slowed down by a factor of 2 to 3.

The code that we write here uses a more 'direct' approach to FP arithmetic, which is variable-oriented rather than stack-oriented (see V1#1). It looks a little more clumsy and is definitely harder to read, but since we want to generate a set of Forth words for general use which should be *fast* more than anything, this is justified.

### Definition of the Problem: Fitting Experimental Data to a Theoretical Equation

Enough of the preliminaries - I should tell you now what exactly we want to do. One of the bread-and-butter problems in experimental science is to extract theoretical parameters from a set of experimental data points, given a theoretical equation that can predict those data points from the parameters.

Example: You measure the time response of a physical system, for instance the voltage across a capacitor C as it is discharged through a resistor R. The time behavior of the voltage versus time looks like:

$$U(t) = U_0 \exp(-t/RC)$$

or, if the voltage does not drop all the way down to zero (e.g. some bias applied),

$$U(t) = U_0 \exp(-t/RC) + U_1$$

In practice, we may have measured a series of points  $U_i$  at times  $t_i$ . Our problem is to get  $U_0$ ,  $U_1$  and  $RC$  from that data. Fig. 1 shows how the data and the 'exact' theoretical curve might look like.

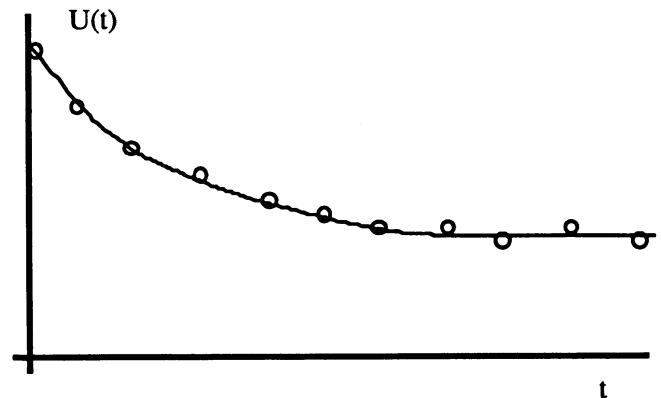


Fig. 1: Fitting a Theoretical Curve to Experimental Data

Of course, for all  $U(t)$  curves with different  $U_0$ ,  $U_1$  and  $RC$ , there is only one that fits the data points best. The quality of the fit is usually checked by summing the squared differences (the 'residuals') between the data points and the theoretical curve. We have to vary the parameters  $U_0$ ,  $U_1$  and  $RC$  in such a way that this sum-of-squares becomes a minimum.

## Iterative Least-Squares Fitting

Let's state the problem in a more general way. We have a function  $y = f(t, a_1, a_2, a_3 \dots a_m)$  that, given certain values for the parameters  $a_1, a_2, a_3 \dots a_m$ , tells us the time dependence of some quantity  $y$  that can be measured. Furthermore we have a set of  $n$  data points  $(t_i, y_i)$ , the  $y$ -values that *are* actually measured at times  $t_i$ . The residual for data point  $i$  is then

$$R_i = f(t_i, a_1, a_2, a_3 \dots a_m) - y_i \quad \text{for } i=0, \dots, n.$$

There exists a variety of techniques that one can use to minimize the sum of the squared residuals in such a case. All of them require that one first estimates initial values for the parameters that are not too far away from reality; this is usually possible. From these initial values one can then compute a better estimate of the parameters, and iterate this process until the fit does not improve anymore.

One rather simple algorithm that solves the fitting problem in such an iterative way is given by T.R. McCalla in his book 'Introduction to Numerical Methods and FORTRAN Programming' (Wiley & Sons, New York 1967). I won't give the details here, since we are mainly interested in how to program such an algorithm in Forth. The only thing we need to know is the final result: a set of linear equations whose solution gives correction terms  $\partial a_k$ . These  $\partial a_k$  have to be added to the initial  $a_k$  to get the new estimate.

The linear equations that one gets look like this:

$$\begin{aligned} c_{11}\partial a_1 + c_{12}\partial a_2 + \dots + c_{1m}\partial a_m &= b_1 \\ c_{21}\partial a_1 + c_{22}\partial a_2 + \dots + c_{2m}\partial a_m &= b_2 \\ &\vdots \\ c_{31}\partial a_1 + c_{32}\partial a_2 + \dots + c_{3m}\partial a_m &= b_3 \end{aligned}$$

**Fig.2: System of Linear Equations**

where the  $c_{ij}$  are coefficients that one calculates from the set of  $n$  data points  $(t_i, y_i)$  and the derivatives of the function  $f(t_i, a_1, a_2, a_3 \dots a_m)$  at each data point with respect to the parameters  $a_k$ . The  $b_i$  contain the residuals.

So the first problem that we have to solve, and this will be plenty for this column, is to solve a system of linear equations like the one given above. In later columns we will build on the basics of floating-point arithmetic that we develop here and end up with a functional curve-fitting program.

## The Gauss Algorithm

A linear equation system like the above is often solved using the Gauss algorithm. Write the coefficients on the left and right hand sides of the equations as a  $m \times m+1$  matrix:

$$\begin{array}{cccc} c_{11} & c_{12} & c_{13} & b_1 \\ c_{21} & c_{22} & c_{23} & b_2 \\ c_{31} & c_{32} & c_{33} & b_3 \\ (3 \text{ by } 4 \text{ in this example}) \end{array}$$

The algorithm then converts this matrix into a triangular matrix:

$$\begin{array}{cccc} c_{11} & c_{12} & c_{13} & b_1 \\ 0 & e_{32} & e_{23} & d_2 \\ 0 & 0 & e_{33} & d_3 \end{array}$$

where the bottom left 'triangle' is equal to zero: multiples of the first row are subtracted from the rows below it until the first column is all zeroes except for the first row, then multiples of the second row are subtracted from the rows below it until the second column is all zeroes except for the first two rows, and so on.

After that procedure is completed, the bottom row has become a simple equation of one variable:

$$e_{33}\partial a_3 = b_3$$

from which  $\partial a_3$  can easily be calculated.  $\partial a_3$  is then substituted into the equation above it and  $\partial a_2$  obtained, and from  $\partial a_3$  and  $\partial a_2$  finally  $\partial a_1$ . This procedure can, of course, be expanded to be used on any number of equations.

The Gauss algorithm is given as a Pascal program (to improve readability) in Listing 1. To code it in Forth we first have to give the problem of data representation a little thought, namely: how are we going to store a matrix?

## Data Representation for Arrays of Floating Point Numbers

The SANE routines work on 80-bit numbers. This is ideal for accurate calculations, but a little expensive as far as storage goes; a  $100 \times 100$  matrix would already occupy 80K. If high precision is not needed, large arrays may be stored as lower precision FP numbers. Single precision uses only 32 bits, less than half of the standard SANE length. Therefore we are going to store matrices as two-dimensional arrays of 32-bit long words that contain single precision real numbers. The MATRIX definition (in the example program in listing 2) is modified from the example released by Creative Solution on the Forth disk. We have separated the DOES> part that calculates the address of a matrix element from its indices and defined it as a separate word, CALC.OFFSET. This was done so that our routine works with any matrix variable whose address is passed on the stack.

You define a matrix with  $r$  rows and  $c$  columns by:

$$r \ c \ \text{MATRIX} \ X$$

When you later execute:

$$i \ j \ X$$

the address of the element in row  $i$  and column  $j$  of matrix  $x$  will be on the stack. When you execute  $0 \ 0 \ X$  (all rows and

columns start with 0), the address of the first element in the matrix will be on the stack. If we want to write a Gauss algorithm routine that works with any matrix of any size, we have to be able to calculate the offset into the matrix from the row and column indices just as the DOES> part of the MATRIX defining word does. In our definitions, `i j addr CALC.OFFSET` leaves on the stack the address of the element at row `i` and column `j` of the matrix whose first element is at `addr`.

The solution of the linear equation system will be stored in an array `z`. For this array we do not need a DOES> part because it is one-dimensional, no need to keep track of row and column lengths here.

### Strategy for Floating Point Calculations Using the SANE Package

The SANE routines expect addresses of floating point numbers on the stack as their parameters (see V1#1). All arithmetic operators are two-address operators, where the first parameter is added to, subtracted from, divided or multiplied into the second parameter. The second parameter is always 80-bit extended precision, while the first one may be any precision. So for any calculation we will transfer the numbers out of the 32-bit variables into 80-bit variables (or add them in etc., if it is convenient), then do all intermediate calculations in 80-bit precision and at the end store the 80-bit result into a 32-bit single precision variable again.

#### The Gauss Algorithm Routine

Listing 2 shows the example program containing the GAUSS routine for solution of linear equation systems of any size. The routine expects on the stack, from bottom to top: the address of a solution vector `z`, which for `n` unknowns has `n` 32-bit words allocated; the address of the `n` (rows) by `n+1` (columns) matrix `X` that contains the coefficients of the linear equation system; and `n`, the number of equations (or unknowns, respectively). The routine first converts the `X` matrix into its triangular form (so `X` is changed upon exit), then proceeds to calculate the values of the unknowns, starting in the bottom row of the matrix and working its way up.

#### The K Function: Extracting the Loop Index 2 Levels Up

The first part of the algorithm has DO..LOOP constructs nested 3 levels deep. The inner loop needs the outermost loop index, and there is no standard word in MacForth that handles this. Therefore we define `: k rp@ 20 + @`; which does this job. (There's also a `k` defined in machine code; see V1#9).

#### Example Program

Our example calculates the solution of the system of equations:

$$\begin{array}{rrrrr} x_1 & + & x_2 & + & x_3 & = & 0 \\ x_1 & - & x_2 & + & 2x_3 & = & 2 \\ 4x_1 & + & x_2 & - & x & = & 4 \end{array}$$

The solution is  $x_1 = 1.2308$ ,  $x_2 = -1.0769$ ,  $x_3 = -0.1538$ . The word `gbm` calculates and prints this solution (it actually calculates `n` times, with `n` on top of the stack, for benchmark purposes).

#### Listing 1: Gaussian algorithm - Pascal example

```
program LinEqu;
type matrix = array[1..10, 1..11] of real;
 column = array[1..10] of real;
var x : matrix; z : column; n, i : integer;

procedure gaussalg (var x : matrix;
 var z : column; n : integer);
var dg, fk, ee : real; i, j, k : integer;
begin
 for i := 1 to n - 1 do
 begin dg := x[i, i];
 for j := i + 1 to n do
 begin fk := x[j, i] / dg;
 for k := i to n + 1 do
 x[j, k] := x[j, k] - fk * x[i, k]
 end
 end
 end;
 for i := 1 to n do z[i] := x[i, n + 1];
 for i := n downto 2 do
 begin dg := x[i, i]; ee := z[i];
 for j := i - 1 downto 1 do
 z[j] := z[j] - ee * x[j, i] / dg
 end;
 for i := 1 to n do z[i] := z[i] / x[i, i]
 end;

begin n := 3;
x[1, 1] := 1; x[1, 2] := 1; x[1, 3] := 1; x[1, 4] := 0;
x[2, 1] := 1; x[2, 2] := -1; x[2, 3] := 2; x[2, 4] := 2;
x[3, 1] := 4; x[3, 2] := 1; x[3, 3] := -1; x[3, 4] := 4;

gaussalg(x, z, n);
for i := 1 to 3 do writeln('z[' , i : 1, ']= ', z[i] : 7 : 4)
end.
```

#### Listing 2: Gaussian algorithm, FORTH example

```
(Floating point primitives)
(This is part of the SANE interface given in MT V1#1; not all of
it is needed here)
hex a9eb w>mt fp68k (package 4)
a9ec w>mt elems68k (package 5)
(extended precision operations)
:f+ 0 fp68k ; : f- 2 fp68k ; : f* 4 fp68k ; : f/ 6 fp68k ;
:x2x e fp68k ; : fneg d fp68k ;

(single to extended operations)
:s+ 1000 fp68k ; : s- 1002 fp68k ; : s2x 100e fp68k ;
:s* 1004 fp68k ; : s/ 1006 fp68k ; : x2s 1010 fp68k ;
```

```

- (long integer to extended operations)
: in+ 2800 fp68k ; : in- 2802 fp68k ;
: in2x 280e fp68k ; : in* 2804 fp68k ;
: in/ 2806 fp68k ; : x2in 2810 fp68k ;
: d2b 9 fp68k ; : b2d b fp68k ;
 (decimal <--> binary conversions)
: float create 10 allot ; : integer create 4 allot ;
: wvar create 2 allot ; (type declarations)
(floating point i/o)
decimal
: numstring create 24 allot ; (decimal display string)
hex 1000000 constant fixdec decimal
(format style control)
variable zzformat
(internal format for conversion routine)
numstring zzs1 (internal conversion string)
: dec. (float\format# --)
 zzformat ! zzformat swap zzs1 b2d
 zzs1 dup w@ 255 > if ."-" else ." " then
 dup 4+ count type (mantissa)
 2+ w@ (get exponent)
 1 w* (convert to 32 bit integer)
 ." E" . ;

(floating point initialization)
: fclear 0 over ! 0 over 4+ ! 0 over 8+ w! drop ;
: sclear 0 swap ! ;

(Matrix Operators)
: calc.offset (row\col\addr -- addr)
 dup>r 4+ @ (#cols) 4* (32-bit)
 rot * (offset to row) swap 4* (32-bit)
 + (offset to element) r> 8+ + (add base addr) ;

: matrix (#rows\#cols --)
 create over, (#rows) dup , (#cols)
 * 4* allot (allot the space for the matrix)
 does> calc.offset ;

(Gauss algorithm for linear equations, definitions)
: k rp@ 20 + @ ;
variable nv variable coeff variable solution
(addresses for storing actual parameters)
float one float -one float zero float two float four
1 one ! -1 -one ! 0 zero ! 2 two ! 4 four !
one one in2x two two in2x -one -one in2x
zero zero in2x four four in2x
float fa1 float fa2 float fa3 float fa4
(define some floating accumulators)
float dg float fk float ee
create z 12 allot 3 4 matrix x
: ztest
 3 0 do i 4* solution @ + fa1 s2x fa1 5 dec. loop cr ;
(setup coefficient matrix for example)
one 0 0 x x2s one 0 1 x x2s one 0 2 x x2s
 zero 0 3 x x2s
one 1 0 x x2s -one 1 1 x x2s two 1 2 x x2s
 two 1 3 x x2s
four 2 0 x x2s one 2 1 x x2s -one 2 2 x x2s
 four 2 3 x x2s
(Gauss algorithm for linear equations)
: gauss (z\'\n | --) nv ! 8- coeff ! solution !

```

```

nv @ 1- 0 do (i-loop)
 i dup coeff @ calc.offset dg s2x (diag elem)
 nv @ i 1+ do (j-loop)
 i j coeff @ calc.offset fk s2x dg fk f/
 nv @ 1+ j do (k-loop)
 k i coeff @ calc.offset fa1 s2x
 fk fa1 f* fa1 fneg (-fk*x[i,k])
 j i coeff @ calc.offset dup fa1 s+
 fa1 swap x2s
 loop
 loop
 loop
nv @ dup 0 do i over coeff @ calc.offset fa1 s2x
 fa1 solution @ i 4* + x2s loop
1 nv @ 1- do
 i dup coeff @ calc.offset dg s2x
 solution @ i 4* + ee s2x dg ee f/
 0 i 1- do i j coeff @ calc.offset fa1 s2x
 ee fa1 f* fa1 fneg
 solution @ i 4* + dup fa1 s+ fa1 swap x2s
 -1 +loop
 -1 +loop
nv @ 0 do solution @ i 4* + fa1 s2x
 i dup coeff @ calc.offset fa1 s/
 fa1 solution @ i 4* + x2s
 loop ;

: soln ." The solution is: " ztest ;

: gbm 0 do z 0 0 x 3 gauss loop soln ;

```



# Forth Forum

## Curve Fitting, Part II

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 12

After last month's refresher on accessing floating point routines from Forth, this column is now going to show you the main part of the curve fitter program.

Let's state our objective again: We have a series of data points (measurements)  $y_i$  at certain time points  $t_i$ . The measured data is supposed to follow some quantitative law, so that we can state a theoretical relationship between time  $t$  and data  $y$ :

$$y = f(t, a_1, a_2, a_3, \dots, a_n)$$

The  $a_i$  are parameters that determine the exact form of the function  $f$ .

Let's assume we have estimated initial values for the parameters  $a_i^0$  somehow so that they are not too far away from the true values. Then we need a method that creates some correction terms  $\partial a_i$ , which when added to the initial values give new, better estimates of the parameters  $a_i'$ :

$$a_i' = a_i^0 + \partial a_i$$

As you could read in last month's column, the  $\partial a_i$  are eventually obtained from the solution of a system of linear equations, and we had defined the word `gauss` to implement the Gauss algorithm that solves such a system.

This month I am going to show you how one sets up the equations (example given for three parameters),

$$c_{11} \partial a_1 + c_{12} \partial a_2 + c_{13} \partial a_3 = b_1$$

$$c_{21} \partial a_1 + c_{22} \partial a_2 + c_{23} \partial a_3 = b_2$$

$$c_{31} \partial a_1 + c_{32} \partial a_2 + c_{33} \partial a_3 = b_3$$

starting from the data and the initial estimate of the function that one wants to fit to it. The full details of the method are given in an appendix to this column.

At this point it is only important to know that the coefficients  $c_{ij}$ , as defined in last month's column, look like the following:

$$c_{ij} = \sum_{k=1}^N \frac{\partial f_k}{\partial a_i} \frac{\partial f_k}{\partial a_j}$$

where  $N$  is the total number of data points and  $\partial f_k / \partial a_i$  is the first derivative of the theoretical function at the time  $t_k$  with respect to the parameter  $a_i$ .

The terms on the right hand side of the equations,  $b_i$ , are:

$$b_i = - \sum_{k=1}^N \frac{\partial f_k}{\partial a_i} R_i$$

$R_i$  are the residuals (as defined last month),

$$R_i = f(t_i, a_1, a_2, a_3, \dots, a_n) - y_i$$

the differences between the theoretical and measured values at the time points  $t_i$ .

Therefore, the curve fitting algorithm will consist of the following major parts:

- One routine that calculates the theoretical function value  $f(t_i)$ , given a set of parameters (this will be the 'model' that you use to fit your data),
- One routine that calculates the derivative of this function with respect to one of the parameters  $a_j$ ,
- Calculation of *all* the derivatives  $\partial f_k / \partial a_i$  with  $k$  ranging from 0 to (# of data points - 1) and  $i$  from 0 to (# of parameters - 1) and saving those derivatives in a matrix,
- Computation of the coefficients  $c_{ij}$  and  $b_i$  (and setting them up in a matrix),
- Solution of the linear equation system thus obtained, giving correction values for the parameters  $a_j$ ,
- Changing the parameters by the correction terms and repetition of the whole algorithm if the change is still larger than some predefined (small) number.

The algorithm is implemented in this month's program example (listing 1). The first part contains some additions to the floating point and Gauss algorithm routines you read last month. One bug had to be corrected in `gauss`, which left a number on the stack that was supposed to be dropped (added *drop* in italics in the listing). The floating point output routine has been slightly modified so that real numbers are always printed with one digit in front of the decimal.

In order to be able to check whether the iteration has been completed or not, we will have to compare real numbers, so first we define (see next page):

## Floating Point Comparison Operator

The SANE package provides a routine for the comparison of two floating point numbers. Four cases are distinguished and the processor status register flags set accordingly:

|           | X | N | Z | V | C |
|-----------|---|---|---|---|---|
| $x < y$   | 1 | 1 | 0 | 0 | 1 |
| $x = y$   | 0 | 0 | 1 | 0 | 0 |
| $x > y$   | 0 | 0 | 0 | 0 | 0 |
| unordered | 0 | 0 | 0 | 1 | 0 |

where the address of  $x$  is on top of the address of  $y$  on the stack.

The operation code for floating point comparison in SANE is 8, so we define

```
hex : f> 8 fp68k @sr 10 and ;
```

for comparing two real numbers on the stack. Bit 5 of the result (the X flag) is **and**-ed out, the reason being that this is the only one that is left unchanged after the SANE trap has been called and the Forth interpreter has taken over again. All the other flags cannot be used, so we won't be able to compare two real numbers for being equal. With 80-bit precision, however, such a test does not make much sense, you will almost never check for equality but rather if a number is less than one other or whether their difference is smaller than some predefined value.

The comparison for 'unorderedness', which sets the V flag, is also trashed by the Forth interpreter. This would only be important if we wanted to test for a NaN (Not a Number), which for instance results from a 0/0 division.

This leaves us with the X flag as the only one usable after the **fp68k** call. It is set when the real number whose address is on top of the stack is less than the number whose address is below it.

## Random Number Generator - Floating Point Version

For initializing the 'data points' as an input to the curve fitting program, we are going to use simulated data from the same function that we are fitting. We add some 'random noise' to it and need a random number generator for that. SANE contains a floating point random number generator, which computes from an 80-bit floating point input a new random 80-bit floating point number in the range  $0 < x < 2^{31}-1$ . We want a number between 0 and 1 (for convenience) so the word **ranf** scales this number after calling the random number routine. **ranf** leaves the address of the result on the stack. For initializing the random number generator with an arbitrary seed, **ranset** is defined.

You will notice one important concept in the definition of **ranf** and **ranset**: both words operate on variables (**rr**, **rs** and **sf**) that are **axed** after the definition is completed. The variables are erased from the vocabulary that way and stay local to the random number routines. This is the way the 'nameless' tokens are created that you read about some issues ago.

The function to be fitted is defined as the word **func**. 50 bytes are reserved for the array **par**, so that at maximum 5 parameters (extended precision) can be used in the function. The data points are stored in the single precision arrays **xdat** and **ydat**. **init\_pars** initializes the parameters to some arbitrary values, in this case  $\text{par}[1] = 1.0$ ,  $\text{par}[2] = 2.0$ , and  $\text{par}[3] = -0.1$ . Calling **func** with these values, **init** then creates a set of simulated data (including random noise).

**deriv** calculates the first derivatives of the function with respect to the parameters from differences,

$$\frac{\partial f_k}{\partial a_i} \approx \frac{(f(t_k, \dots, a_i + \Delta a, \dots) - f(t_k, \dots, a_i - \Delta a, \dots)) / (2 \Delta a)}{}$$

This definition, too, uses the local variables **da1**, **da2**, **da3** and **da4**, which are **axed** afterwards.

**make\_derivmat** computes all necessary derivative values and sets them up in a single precision matrix. This matrix is then used by the word **make\_resmat** to compute the sums that make up the coefficient matrix. For the right-hand-side coefficients one also needs the residuals, which are calculated and stored into a matrix by **residuals**.

One iteration of the curve fitting process is done by the word **one\_iter**, which sets up the derivative matrix and computes the residuals by calling the appropriate words, prints the sum of error squares (so you can check the quality of the fit) then sets up the coefficient matrix and calls **gauss** to solve the linear equation system. The solution is stored in the array **delta**; these are the correction terms that have to be added to the parameters to get improved estimates. **new\_pars** does this correction and prints out the values of the new parameters, leaving false (zero) on the stack if any parameter has changed by more than one part in  $10^{-5}$ .

The actual curve fitter, **nlscfit**, then loops through the iteration until the best fit is obtained.

You can check the fitting process by calling **init** first and then, for instance, setting  $\text{par}[1] = 2.0$ ,  $\text{par}[2] = 1.0$ ,  $\text{par}[3] = -0.05$ :

```
two par x2x one par 10+ x2x
two par 20 + f/
```

and start **nlscfit**. This will bring you back close to the simulated values,  $\text{par}[1] = 1.0$ ,  $\text{par}[2] = 2.0$ , and  $\text{par}[3] = -0.1$  in about 4 to 5 iterations. The fitted values will not be exactly equal to the simulated ones because of the random noise added to the data.

So far, this is only a skeleton of a curve fitting program because we cannot input floating point numbers manually or from the clipboard (e.g. it would be nice to transfer data from a spreadsheet); also a graphical output will be needed to display the data points and the fitted curve. Next column will deal with those problems.

### Listing 1: Non-linear Least Squares Curve Fitting Routine

(© 1985 J.Langowski for MacTutor)

(Note that this is not stand-alone but needs some definitions from last month's example. Only the changed parts are printed here)

hex

```
:f> 8 fp68k @sr 10 and ; : fabs f fp68k ;
:lnx 0 elems68k ; : log2x 2 elems68k ;
:ln1x 4 elems68k ; : log21x 6 elems68k ;
:expx 8 elems68k ; : exp2x a elems68k ;
:exp1x c elems68k ; : exp21x e elems68k ;
:x^i 8010 elems68k ; : x^y 8012 elems68k ;
:compoundx c014 elems68k ;
:annuityx c016 elems68k ;
:sinx 18 elems68k ; : cosx 1a elems68k ;
:tanx 1c elems68k ; : atanx 1e elems68k ;
:randomx 20 elems68k ; decimal
```

:dec. (float\format# --)

```
zzformat ! zzformat swap zzs1 b2d
zzs1 dup w@ 255 > if ." -" else ." " then
dup 4+ count over 1 type ." "
swap 1+ swap 1- type (mantissa)
2+ w@ (get exponent)
1 w* zzformat @ + 1-
." E" . ;
```

(define constants)

```
float one float -one float zero float two float four
1 sp@ one in2x drop -1 sp@ -one in2x drop
0 sp@ zero in2x drop
2 sp@ two in2x drop 4 sp@ four in2x drop
(define some floating accumulators)
float fa1 float fa2 float fa3 float fa4
```

(Gauss algorithm for linear equations)

```
float dg float fk float ee
variable nv variable coeff variable solution
(addresses for storing actual parameters)
:gauss (z\ \n | --) nv ! 8- coeff ! solution !
nv @ 1- 0 do (i-loop)
i dup coeff @ calc.offset dg s2x (diag elem)
nv @ i 1+ do (j-loop)
i j coeff @ calc.offset fk s2x dg fk f/
nv @ 1+ j do (k-loop)
k i coeff @ calc.offset fa1 s2x
fk fa1 f* fa1 fneg (-fk*x[i,k])
j i coeff @ calc.offset dup fa1 s+
fa1 swap x2s
loop
loop
```

```
loop
nv @ dup 0 do i over coeff @ calc.offset fa1 s2x
fa1 solution @ i 4* + x2s loop drop
1 nv @ 1- do
i dup coeff @ calc.offset dg s2x
solution @ i 4* + ee s2x dg ee f/
0 i 1- do j coeff @ calc.offset fa1 s2x
ee fa1 f* fa1 fneg
solution @ i 4* + dup fa1 s+ fa1 swap x2s
-1 +loop
-1 +loop
nv @ 0 do solution @ i 4* + fa1 s2x
i dup coeff @ calc.offset fa1 s/
fa1 solution @ i 4* + x2s
loop ;
```

(declarations for curve fitter)

```
create ydat 400 allot create xdat 400 allot
create residues 400 allot
100 10 matrix derivmat 10 11 matrix resmat
3 constant npars 10 constant npts
create par 50 allot create delta 20 allot
float eps float errsum
1 sp@ eps in2x drop 10000 sp@ eps in/ drop
float onehundred 100 sp@ onehundred in2x drop
float ten 10 sp@ ten in2x drop
(define function)
:func (x -- f[x] = par[1] + par[2] * exp[par[3]*x])
par 20 + over f* dup expx
par 10 + over f* par over f+ ;
:test 10 0 do i sp@ fa1 in2x . 2 spaces
fa1 func 10 dec. cr loop ;
:>fa1 fa1 s2x ;
:init_pars
one par x2x two par 10+ x2x
-one par 20 + x2x ten par 20 + f/ ;
init_pars
```

(derivative, matrix of derivs)

```
float da1 float da2 float da3 float da4 (local vars)
:deriv (par \ x -- d-func/d-par at x)
dup da1 x2x da2 x2x dup da4 x2x eps da4 f*
da4 over f+ da2 func da3 x2x
da4 over 2dup f- f- da1 func da3 f-
da4 da3 f/ two da3 f/ da4 swap f+ da3 ;
axe da1 axe da2 axe da3 axe da4
```

:make\_derivmat

```
npts 0 do npars 0 do
xdat j 4* + >fa1
par i 10 * + fa1 deriv j i derivmat x2s
loop loop ;
```

(calculate residuals)

```
:residuals
zero errsum x2x
npts 0 do
xdat i 4* + >fa1 fa1 func ydat i 4* + swap s-
fa1 residues i 4* + x2s fa1 dup f* fa1 errsum f+
loop ;
:resid
npts 0 do residues i 4* + >fa1 fa1 7 dec. cr loop ;
```



```

(make matrix of residuals)
make_resmat
npars 0 do npars 0 do zero fa1 x2x
 npts 0 do
 i k derivmat fa2 s2x i j derivmat fa2 s*
 fa2 fa1 f+ loop
 fa1 i j resmat x2s fa1 j i resmat x2s
 loop loop
npars 0 do zero fa1 x2x
 npts 0 do
 i j derivmat fa2 s2x residues i 4* + fa2 s*
 fa2 fa1 f- loop
 fa1 i npars resmat x2s loop ;

(calculate correction terms)
:one_iter
make_derivmat residuals
."sum of error squares: " errsum 7 dec. cr
make_resmat delta 0 0 resmat npars gauss ;

:new_pars 16 (true if no significant changes)
npars 0 do par i 10 * +
 delta i 4* + over s+
 ."par[" i . ."] = " dup 7 dec. cr
 delta i 4* + fa1 s2x fa1 f/
 fa1 fabs eps fa1 f> and loop ;

(ranf, initialize data matrices)
float rr float rs float sf (local to ranf)
1 31 scale 1 - sp@ sf in2x drop
:ranset rr x2x ;
:ranf rr randomx rr rs x2x sf rs f/ rs ;
axe rr axe rs axe sf
12345678 sp@ fa1 in2x drop fa1 ranset

80 ' npts !
: init npts 0 do i sp@ fa1 in2x 4*
 xdat over + fa1 swap x2s
 ydat over + fa1 func ranf fa2 x2x
 ten fa2 f/ fa2 over f+ swap x2s
 i . xdat over + >fa1 fa1 7 dec. 2 spaces
 ydat + >fa1 fa1 7 dec. cr loop ;

(print matrices for debugging)
:.dmat
npts 0 do
 npars 0 do j i derivmat >fa1 fa1 5 dec. loop
 cr loop ;
:.rmat
npars 0 do
 npars 1+ 0 do j i resmat >fa1 fa1 5 dec. loop
 cr loop ;

(nonlinear fit, core routine)
:nlsqfit cr begin one_iter cr new_pars cr until ;

```

### Appendix: Theoretical Background of the Curve Fitting Routine

We want to determine the values of the  $a_i$  in such a way that the differences between the theoretical function and the

measured  $y_k$  values at times  $t_k$  become a minimum. These differences are called the *residuals*  $r_k$ :

$$r_k = f(t_k, a_1, a_2, a_3, \dots, a_n) - y_k$$

and one usually tries to minimize the sum of the squared residuals of all data points.

Let's assume  $r_i$  are the 'true' residuals that one obtains with the exact  $a_i$  values. If we estimate the parameters by some initial values  $a_i^\circ$ , then 'computed' residuals

$$R_k = f(t_k, a_1^\circ, a_2^\circ, a_3^\circ, \dots, a_n^\circ) - y_k$$

can be calculated, which are usually larger than the true ones. To get a correction term that brings the  $a_i^\circ$  closer to the 'true'  $a_i$ , one now linearly expands the function  $f$  around the estimated value:

$$\begin{aligned}
 f(t_k, a_1, a_2, \dots, a_n) \approx & f(t_k, a_1^\circ, a_2^\circ, \dots, a_n^\circ) + \frac{\partial f_k}{\partial a_1} (a_1 - a_1^\circ) \\
 & + \frac{\partial f_k}{\partial a_2} (a_2 - a_2^\circ) \\
 & \dots \dots \\
 & + \frac{\partial f_k}{\partial a_n} (a_n - a_n^\circ)
 \end{aligned}$$

The differences,  $(a_i - a_i^\circ)$ , are denoted by  $\partial a_i$ , now we can write

$$\begin{aligned}
 f(t_k, a_1, a_2, \dots, a_n) - y_k \approx & f(t_k, a_1^\circ, a_2^\circ, \dots, a_n^\circ) - y_k \\
 & + \frac{\partial f_k}{\partial a_1} \partial a_1 + \frac{\partial f_k}{\partial a_2} \partial a_2 \\
 & \dots \dots \\
 & + \frac{\partial f_k}{\partial a_n} \partial a_n
 \end{aligned}$$

which gives us a relationship between the true and the computed residuals

$$\begin{aligned}
 r_k \approx R_k + \frac{\partial f_k}{\partial a_1} \partial a_1 + \frac{\partial f_k}{\partial a_2} \partial a_2 \\
 \dots \dots \\
 + \frac{\partial f_k}{\partial a_n} \partial a_n .
 \end{aligned}$$

It is the sum of the squares of the true residuals ( $N$  being the number of data points) that has to be minimized with respect to changes in  $\partial a_i$ , this means all the derivatives  $\partial Q / \partial (\partial a_i)$  have to be zero simultaneously. When you evaluate the expressions for the  $\partial Q / \partial (\partial a_i)$  and set them to zero, you arrive at the equation system that was described in the main article.

$$Q = \sum_{k=1}^N r_k^2 = \sum_{k=1}^N \left[ R_k + \frac{\partial f_k}{\partial a_1} + \frac{\partial f_k}{\partial a_2} + \dots + \frac{\partial f_k}{\partial a_n} \right]^2$$



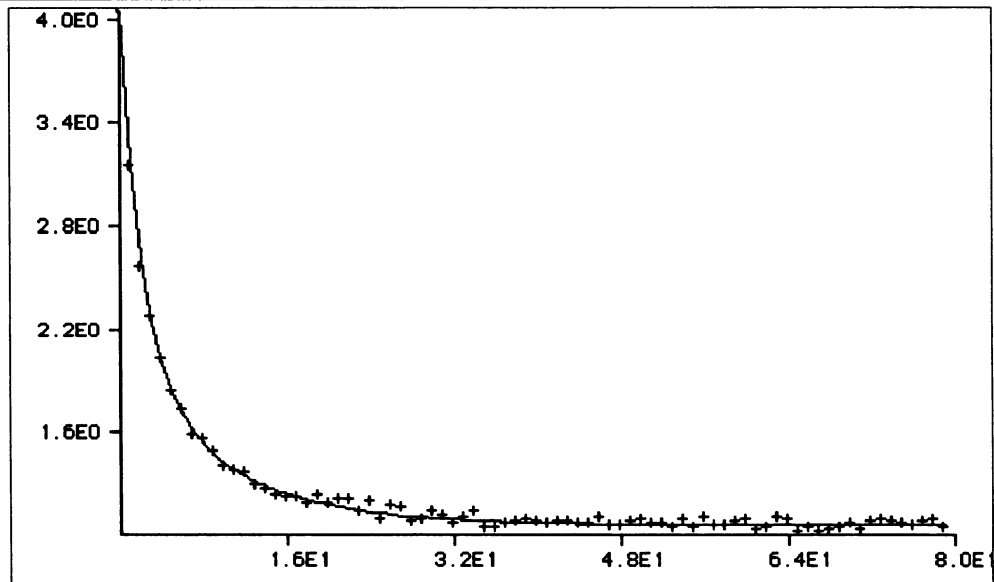
# Forth Forum

## Curve Fitting, Part III

Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 13

Last time I promised you to add some input/output to the curve fitter program so that it becomes a useful application. We'll do this here today.

A lot of basic graphic routines which you would have to write yourself on other machines are included in the Quickdraw ROM. This makes life very easy; we can use the screen like we would use a plotter (of course, we can do much better than that, but this is what we are dealing with today). What is not included in the ROM, of course, are utilities that plot coordinate axes, draw lines through pairs of x/y points, add symbols, etc.



Most computer systems that are used for numerical calculation purposes contain a set of standard plotting routines, mostly callable from FORTRAN, that will help you generating a 'nice' graphical output with not too much effort. This column gives you some similar routines in FORTH; I have tried to stick closely to the Calcomp plotting routines, since they are something of a standard.

Our objective is to produce something like the curve in Fig. 1. The data points that the model is fitted to will be displayed using some symbols, while the fitted curve is plotted as a smooth line. X- and Y-axes are added to the plot.

The numbers that are to be plotted will be given in floating point format (in this case, single precision arrays as defined previously). So first of all we have to know how to scale the data to the integer values of the bit map that we are going to plot to.

The Calcomp convention is, for an array of N floating point numbers, to store the minimum of these numbers in cell N+1 of the array and the difference between the minimum and the maximum in cell N+2. These two numbers are then rounded to one significant figure. Given this information, one can write routines that automatically draw an axis that spans the range of the data values.

The rounding to one significant figure is done by the word `next.int`, which given the address of an extended number on the stack returns with the address of the rounded number (which is a local variable of `next.int`).

`fscale` finds the minimum and maximum values of an array and stores the scaling numbers above the last data point after rounding them to one significant digit.

`axis` and `yaxis` draw coordinate axes in x- and y-direction. Input parameters are `(#ticks\length\array\npts)`, `#ticks` gives the number of ticks to be drawn (to the bottom of the x- and to the left of the y-axis), `length` is the total axis length in points, `array` the address of the scaled data array and `npts` the number of points in `array`.

Given two scaled arrays, `line` draws a line through the (x y) pairs defined by the two arrays. The symbol defined by the global `symbol` is plotted at each of the data points. Symbols could be e.g. '+' or '\*'. Setting `symbol` to zero will draw no symbols (how could you have guessed). Depending on the setting of the global flag `connecting`, the symbols are either connected by a line or not.

For all the plotting routines `cartesian` has to be switched on and the origin to be defined within the output window by `xyoffset`. `init.plot` is used to do this job.

The main curve fitter program now consists of the following:

1. The data arrays `xdat` and `ydat` are initialized with the simulated 'experimental data' (`init`). In this month's example, we use a sum of two exponentials as our model; this gives us five parameters to fit.

2. A routine is called in which one can deliberately change the values of the parameters (for this we need floating

point input, see below), so that one can see how the correct curve is fitted through the data starting from wrong parameters. One may also change the total number of parameters to be fitted; for instance, fitting three parameters only will force a single exponential fit to the data points. In this case, parameters 4 and 5 will have to be set to zero.

3. The main loop calculates theoretical function values from **xdat** and the parameters and stores them in **zdat**. The data arrays are scaled and the plot displayed (like in Fig. 1); then one iteration of the fitting routine is taken, the parameters changed accordingly, displayed and the process repeated until parameter changes are below 1 part in  $10^4$ .

### Floating Point Input

As I mentioned, the program would not be very useful without floating point input routines. In order to be independent of any particular Forth implementation, I am including a simple floating point input routine here which accepts a string and converts it to the decimal string format used by the SANE conversion routines.

Any such routine would mainly consist of taking successive characters from the input string, generating a decimal mantissa and keeping track of the number of digits behind the decimal point, then looking for an 'e' or 'E' to indicate an exponent and converting the exponent finally. In that aspect, the routine written here is very similar to the MacForth floating input routine. There is one difference, in that numbers *without* an exponent will be accepted and converted to floating point numbers, too. Therefore, you won't be able to use this routine as an extension of the Forth interpreter, as MacForth level 2 does. However when entering a lot of data, it can be very tedious always having to type an exponent.

**fnumber** takes a string as its input parameter and leaves on the stack:

( **addr of float** **true** ) if a valid floating point number was read from the string,  
( **false** ) if the conversion was not successful.

In the latter case, an error message is printed.

**input.float** reads a string from the keyboard into **pad**, then calls **fnumber** to convert it.

With the additions from this column, the curve fitter should finally be a utility that is useful to you (in case you have any curves to fit). Changing the function to be fitted is easy, and you might even install a **make** switch for vectored execution (V1#7) so that you can easily switch between different functions within one program.

Data input still has to be done manually, number by number. However, **input.float** may easily be extended to read input from a file. To transfer data to/from other

applications through the scrap requires some more work; I'll deal with that problem soon.

### Re: Finding Object Code of Unnamed Tokens

The procedure to decompile the object code of an axed token is the following (V1#2):

Convert the token to an absolute address, using **token>addr**. If the word contained there is **\$4e4f** (TRAP \$F), the next words will be Forth code. Start decompiling at the following word. Of course, different versions of MacForth will give different addresses for the individual words due to different dictionary arrangements; but this procedure should work for any version of Level 1 or Level 2.

### Re: MacModula Floating Point

Shortly after my comment on errors in MacModula 2's 32-bit floating point arithmetic, I received a letter from the author of those routines, Daan Strebe:

"...A few weeks ago I followed several bug reports to find two fundamental errors in the floating-point routines, one a conceptual error in the rounding, and the other a misunderstanding about the 68000 processor instruction set. These two errors affected the multiply routine most, although the others were somewhat affected also. I revamped those routines and then ran a complex set of comprehensive tests, and the results allow me to state with a margin of confidence that the floating point in the latest rev is now not only slightly faster than it was, but also that the only errors in the basic routines (not necessarily including those in the math library) are from rounding. The rounding as it is now implemented yields 3 downward, 4 upward, and 1 non-round per 8 random floating-point operations. This should be satisfactory for most users; full IEEE rounding would considerably decrease the speed. I believe the compromise was successful..."

So everything should be fixed now. Let's hope that the new update will be distributed soon (it probably is when this goes to the printer's).

#### Listing 1: Plotting Routines and Floating Point Input for the Curve Fitter Program

( Additions to the curve fitting program )  
( for graphical output and floating point input. )  
( © 1985 J. Langowski for MacTutor )  
( Again, only the parts that have been changed )  
( or added with respect to the last two columns )  
( are printed here )

```
: s> 1008 fp68k @sr 10 and ;
```

```
: numstring create 24 allot ;
numstring zzs1 (internal conversion string)
```

```

: dec. (float\format# --)
 zzformat ! zzformat swap zzs1 b2d
 zzs1 dup w@ 255 > if ."-" else ." " then
 dup 4+ count over 1 type ."."
 swap 1+ swap 1- type (mantissa)
 2+ w@ (get exponent)
 1 w* zzformat @ + 1-
 ." E" 0.r ;

create xdat 400 allot create ydat 400 allot
create zdat 400 allot create residues 400 allot
100 10 matrix derivmat 10 11 matrix resmat
5 constant npars 80 constant npts
create par 5 10 * allot
create delta 5 4* allot

float logten ten logten x2x logten lnx
: log10x dup lnx logten swap f/ ;

(define function) (090485 jl)
float temp (local to func)

func (x -- f[x] = par[1] + par[2] * exp[par[3]*x]
 + par[4] * exp[par[5]*x])

dup temp x2x
par 40 + temp f* temp expx par 30 + temp f*
par 20 + over f* dup expx par 10 + over f*
par over f+ temp over f+ ;
axe temp

:>fa1 fa1 s2x ;
: init_pars one par x2x two par 10+ x2x
 -one par 20 + x2x two par 20 + f/
 one par 30 + x2x
 -one par 40 + x2x ten par 40 + f/ ;
init_pars

: one_iter
make_derivmat residuals
make_resmat delta 0 0 resmat npars gauss ;

: new_pars 16 (true if no significant changes)
npars 0 do par i 10 * +
delta i 4* + over s+
delta i 4* + fa1 s2x fa1 f/
fa1 fabs eps fa1 f> and loop ;

80 ' npts ! 5 ' npars !

: init (initialize data arrays)
npts 0 do i sp@ fa1 in2x 4*
xdat over + fa1 swap x2s
ydat over + fa1 func ranf fa2 x2x
ten fa2 f/ fa2 over f+ swap x2s
drop loop ;

(plotting routines) (092385 jl)
float 1/2 1 sp@ 1/2 in2x drop 2 sp@ 1/2 in/ drop
float small ten small x2x -200 sp@ fa1 in2x drop
fa1 small x^y
(anything smaller than small will be zero)
float sc.aux float sc.exp
: next.int (float -- rounded to 1 dec. place)

```

```

dup small f>
if dup sc.aux x2x sc.aux dup fabs dup log10x
dup 1/2 swap f- frti
ten sc.exp x2x sc.aux sc.exp x^y sc.aux x2x
sc.exp sc.aux f/ sc.aux frti sc.exp sc.aux f*
sc.aux
else drop zero then ;

float xlow float xhi float sc.factor
: fscale
(array \ n -- | start, scale -> array[n+1..n+2])
over xlow s2x over xhi s2x
over over 1 do
dup i 4* + dup xlow s> not if dup xlow s2x then
dup xhi s> if xhi s2x else drop then loop
xlow xhi f-
over 4* + xlow next.int swap x2s
1+ 4* + xhi next.int swap x2s ;

: .fscale (array \ n --)
4* + dup fa1 s2x ." min = " fa1 7 dec.
4+ fa1 s2x ." , scale = " fa1 7 dec. cr ;

(xtick, ytick) (092385 jl)

: xtick (rx x --) dup 0 move.to
dup -5 draw.to dup 15 - -16 move.to
get.textsize >r 9 textsize swap 2 dec. r> textsize
0 move.to ;

: ytick (ry y --) -45 over 4- move.to
get.textsize >r 9 textsize swap 2 dec. r> textsize
0 over move.to -5 over draw.to
0 over 6- move.to 0 swap draw.to ;

(xaxis) (092385 jl)
float start float del
: xaxis
(#ticks\length\array\npts -- | starts at origin)
0 0 move.to
4* + dup start s2x 4+ delta s2x
over / (#ticks\length/tick --)
over 0 do dup i 1+ * dup 0 draw.to
delta fa1 x2x i 1+ sp@ fa1 in* drop
3 pick sp@ fa1 in/ drop
start fa1 f+ fa1 swap xtick loop
drop drop ;

(yaxis) (092385 jl)
: yaxis
(#ticks\length\array\npts -- | starts at origin)
0 0 move.to
4* + dup start s2x 4+ delta s2x
over / (#ticks\length/tick --)
over 0 do dup i 1+ * 0 over draw.to
delta fa1 x2x i 1+ sp@ fa1 in* drop
3 pick sp@ fa1 in/ drop
start fa1 f+ fa1 swap ytick loop
drop drop ;

(line, variables) (092585 jl)

```

```

variable xline.length variable yline.length
variable xpos variable ypos variable symbol
variable connecting connecting off
float xstart float ystart float xdel float ydel

```

```

(line) (092585 jl)
: line (xarray\yarray\npts\length\length --)
 yline.length ! xline.length ! 0 0 move.to
 over over 4* + dup ystart s2x 4+ ydel s2x
 3 pick over 4* + dup xstart s2x 4+ xdel s2x
 0 do over i 4* + fa1 s2x xstart fa1 f- xdel fa1 f/
 dup i 4* + fa2 s2x ystart fa2 f- ydel fa2 f/
 xline.length fa1 in* fa1 xpos x2in
 yline.length fa2 in* fa2 ypos x2in
 xpos @ ypos @ over over
 connecting @ if draw.to else move.to then
 -4 -4 rmove symbol @ emit move.to
 loop drop drop ;

```

```

(calc.theor, disp.theor, disp.data, scale.all)
(092585 jl)
: calc.theor

```

```

 npts 0 do xdat i 4* + fa1 s2x fa1 func
 zdat i 4* + x2s loop ;

```

```

: disp.theor
 0 symbol ! connecting on
 xdat zdat npts 400 250 line ;

```

```

: disp.data
 43 symbol ! connecting off
 xdat ydat npts 400 250 line ;

```

```

: scale.all
 xdat npts fscale ydat npts fscale
 ydat npts 4* + @ zdat npts 4* + !
 ydat npts 1+ 4* + @ zdat npts 1+ 4* + ! ;

```

```

(disp.axes) (092585 jl)

```

```

: disp.axes
 5 400 xdat npts xaxis 5 250 ydat npts yaxis ;
: init.plot
 page 50 255 xyoffset cartesian on ;

```

```

(floating point input) (092385 jl)

```

```

: fsign zzs1 ; : fexpo zzs1 2+ ;
: fmant zzs1 4+ ; variable frac float float.out
variable decimals
: input.sign 0 fsign !
 dup c@ case 45 of -1 fsign w! 1+ endof
 43 of 1+ endof endcase ;
: input.mantissa 0 decimals ! 0 frac !
 fmant 21 + fmant 1+ do i fmant 1+ - fmant c!
 dup c@ dup
 case 48 57 range.of ic! frac @ decimals +!
 1+ 1 endof
 46 of drop 1 frac ! 1+ 0 endof
 20 swap endcase +loop drop ;
: input.exponent
 dup c@ dup bl = not
 if dup 69 = swap 101 = or not
 if drop 0
 else dup 1+ c@ 43 = 1 and +
 number decimals @ - fexpo w! -1

```

```

then
else decimals @ -1 * fexpo w! -1 2drop
then ;

```

```

: fnumber zzs1 24 blanks
 input.sign input.mantissa input.exponent
 if fsign float.out d2b float.out -1
 else ." floating input error!" cr 0 then ;

```

```

: input.float pad 22 blanks pad 22 expect pad fnumber ;

```

```

: get.pars
 npars 0 do
 begin cr ." par[" i 1+ . ."] = " input.float until
 par i 10 * + x2x loop
 cr begin cr ." total # of parameters to be fitted "
 5 input.number until
 ' npars ! cr ;

```

```

: .pars npars 0 do
 ." par[" i 1+ . ."] = " i 10 * par + 7 dec. cr loop ;

```

```

(main curve fitter program) (092685 jl)

```

```

: fit.curve page
 init_pars ." initializing data arrays..." cr init
 5 ' npars ! init.plot get.pars .pars
 ." fitting " npars . ." parameters to "
 npts . ." data points "
 calc.theor scale.all page
 disp.data disp.axes disp.theor
 begin one_iter new_pars not while
 page .pars
 calc.theor page disp.data disp.axes disp.theor
 repeat
 300 300 move.to ;

```



# Fortran's World

## Random Generator Shows Off MacFortran



Mark McBride  
Oxford, OH  
MacTutor Vol. 1 No. 10

Having used the Macintosh for over a year in my routine daily work of word processing and telecommunicating and for some minimal programming in Pascal and assembly, I decided it was time to consider expanding the activities I do on the Mac. As an economist, most of my computer work focuses on complex statistical analysis on the mainframe computers using SAS, SPSS, and Shazam, as well as custom Fortran programs. Therefore, I began investigating ways to convert the custom mainframe Fortran programs to the Mac.

This column will cover three topics of interest to Mac users wanting to program in Fortran or convert Fortran programs from other machines. First, tutorials on how to implement Mac features from within Fortran programs. Second, strategies for converting Fortran programs to the Mac. And third, combining the first two, creating custom Mac versions of your mainframe programs which take maximum advantage of the Mac's user interface with the least amount of conversion effort.

### Which Fortran?

In beginning my quest, I laid down four criteria for the conversion of Fortran programs to the Mac. First, I should be able to easily import the source code from the mainframe via the modem. Second, I should be able to compile and run the existing code on the Mac with a minimum of conversion problems. Third, I should be able to redesign parts of the code to have Mac-like features, as I deemed necessary. And finally, I should be able to create self-contained applications.

### MacFortran by Absoft

In evaluating Fortran systems to

```
System tick count at start is 1521110

The average number of responses in sample of size 60 after 1000 trials is

For die value 1 the average number of responses was 9.99200
For die value 2 the average number of responses was 10.05600
For die value 3 the average number of responses was 10.19400
For die value 4 the average number of responses was 9.97400
For die value 5 the average number of responses was 10.02900
For die value 6 the average number of responses was 9.75500

System tickcount at end is 1526579

The sampling took 91.00 seconds

Press Return to quit
```

Fig. 1 Screen output of this month's Fortran program

meet these criteria, I discovered that only MacFortran by Absoft appeared to meet all four criteria. The Absoft system (version 2.0a; version 2.1 is due for release this month) consists of Apple's Edit program (a general purpose text editor used in the MDS and Consulair C systems), a Fortran 77 compiler, a linker, an interactive debugger, and a library manager. Only the linker appears to have some deficiencies (more on that below). MacFortran compiles into 68000 machine code. The compiler has many options, including an option for compiling Fortran 66 source code. Probably one of the most interesting features of the compiler is the ability to generate 68000 assembly source code which can be used with an assembler to fine tune the code. (This is also useful for studying how compilers produce native code.) MacFortran's desirable language features include structured programming constructs: **if then**

**else, block if, case, etc.** The interactive debugger is almost as easy to use and as powerful as the debugger in MacPascal. This makes debugging relatively easy in MacFortran. While the documentation is thin in places, the system is fairly easy to get up and running and access to the toolbox is well documented. (Version 2.1 has an extra 100 pages of toolbox docs included.)

### Using MacFortran

MacFortran requires two drives to use it effectively. I keep Edit, the compiler, the debugger, and the MacFortran overlay files on the internal drive along with a stripped down Mac system file and an Imagewriter file. This leaves about 35k of free space. I have removed the MacFortran overlay file -- f77003.fc-- from my work disk. This overlay file is necessary only if you are going to generate assembly

source code during compiles. In the external drive, I keep commonly used subroutine code files, source files, the linker, and the library manager. The external disk does not contain any Mac system files. This arrangement seems to provide a workable environment for developing programs. (Don't you agree the Mac should have had the 800K drives it was designed for? -Ed)

## Converting a Fortran Program

For a first attempt at converting programs, I choose a relatively simple Fortran source code: an economic simulation of a small third world country (Fredonia) developed by a colleague. The Fredonia simulation is about 1500 lines of code which is essentially a large DO LOOP over straight line code. Only 3 short subroutines exist within the source code. MacTerminal provided easy downloading of the code from the HP 3000 minicomputer via text capture. The Edit program read the MacTerminal file, allowing me to remove all the extraneous information from the download.

Before compiling the source code, I checked the file I/O conventions used in the source code against those available under MacFortran. MacFortran handles the file I/O specifications of Fortran 77. However, there are a few items worth noting. First, while writing output to the printer via unit 6 is possible, data input with read statements using unit 5 are not available in MacFortran. Second, the file unit identifier \* is connected to the I/O devices defined by units 5 and 6. The \* unit identifier can be redirected to the keyboard for reads and the screen for writes with a compiler option. Third, the file unit identifier 9 is connected to the keyboard for reads and the screen for writes. Fourth, other files read from or written to need to have their OPEN statements checked against the syntax that MacFortran uses for open statements. If the mainframe program used JCL statements to assign file I/O, then the ported program will need OPEN statements added to establish the file preconnections. A final word of caution

concerns writing to the printer via unit 6. MacFortran will spool your output to a temporary file. This temporary file will be sent to the printer when your program ends by a subroutine called spool. To avoid out of memory errors at runtime (error=64), include the following subroutine in your source:

```
subroutine dum
integer*2 ary(1000)
common ary
return
end
```

This forces MacFortran to reserve enough memory to load the spool subroutine.

The Fredonia source file compiled with only one error on the first pass. This error was caused by the absence of a Fortran subprogram specific to the HP 3000 which generates random numbers. The ensuing search for a random number generator allowed me to explore some of the intricacies of the MacFortran system.

## Implementing Random Numbers

A search of the MacFortran manual revealed that access to the toolbox included the random number toolbox routine which is part of Quickdraw (see *Inside Mac* for a description of random). The toolbox random routine can be seeded in a system global variable, RndSeed, in order to generate different random sequences. When Quickdraw is initialized, RndSeed is set to zero.

Given that the location of RndSeed is known via an offset to a pointer contained in register A5, direct access to the seed location from Fortran was not possible. This is because the present system does not support access to the quickdraw globals. However, Absoft states version 2.1 includes access to system global variables, as well as RAM based packages, which are not currently supported. I decided to write a short assembly language routine to seed the random number generator.

This turns out to be more difficult than it sounds. Apparently MacFortran

also uses the A5 register internally, forcing Absoft to relocate the Mac pointer contained in A5 that assembly programmers have learned to love. Absoft states that the Mac pointer is located at -4(A0), but I still haven't gotten the assembly based seeding to work yet. This is not to be critical of Absoft, who have been extremely patient and helpful in answering the duffer questions of a neophyte assembly programmer. I decided to wait on version 2.1 to access globals directly from Fortran. Others who want to call external assembly language subprograms would be well advised to contact Absoft to find the best solution for their particular needs, and in particular, making sure they have version 2.1. I will report a general solution as soon as I figure/ find it out.

The next solution tried was to implement a random number generator in Fortran. The routine I selected is given in Rand.for shown in listing 2. This random number function returns a real value between zero and one. The random sequence can be reseeded on any call or a fixed sequence can be maintained for program debugging purposes.

Examination of the random number function source code reveals two interesting features. One is the use of the Save statement. A subprogram in MacFortran does not save local variable values between successive calls of the program. The random number generator needs to retain local variable values between calls and does so with the Save statement.

## ToolBox Support

Second, I use the Macintosh toolbox traps TickCount and BitXor. MacFortran uses a general purpose subprogram called toolbox to handle toolbox traps. Toobx is defined as integer\*4 and can be used as either a subroutine call or as a function call. MacFortran keeps track of which is being done based upon which toolbox routine is called.

To make a toolbox call in MacFortran the program defines and assigns values to the necessary

variables of information to be passed or received. This information is passed along with the name of the toolbox routine by a call to the toolbox subprogram (Note: this is very similar to the inLine procedure in MacPascal).

Absoft has provided include files that handle the toolbox name and parameter declarations. Absoft has also provided example source code files and good documentation on accessing the toolbox from Fortran.

[The way this toolbox calling is implemented is non-standard. A source file called `toolbox.par` contains a list of trap names equated to integers from 0 to 500 or so. The trap name is set equal to this integer index in your program by either including this file of definitions, or using the parameter statement to assign the integer value to the trap name. Apparently the subroutine `toolbox.sub` uses the integer value as an index into its own table of glue routines, where it obtains the real trap address and executes the toolbox call. The source code to `toolbox.sub` is *not* provided. Since there is no obvious connection between the integer values in the file `toolbox.par` and the trap address, and since the trap names are not in any kind of order, it is a bit difficult to wade through the 500 assignments looking for the trap call you're interested in. The trap names *are* standard so by including the parameter file you don't have to bother trying to figure out how Absoft happened to assign those integer values to each trap name! And a problem with the toolbox definitions is that in the `toolbox.par` file included in version 2.0, many of the trap calls are unimplemented with a comment saying "...for future implementation...". We have already mentioned the fact that RAM based packages and access to the quickdraw globals is missing in this version. We have requested a copy of version 2.1 to see if the toolbox is more fully implemented, as we expect. We will report on version 2.1 in the next issue, including a review of the `toolbox.par` file to see how many trap calls are implemented. -Ed.]

## Using the Rand Function

Listing 1 provides the source file **Random Test**.for which checks the accuracy and speed in using the Tausworthe random number generator. This program does not create or use any special Mac features, such as windows or menus. The program uses the default tty like environment Absoft has set up. This environment is useful when first porting over a Fortran program to ensure that the compiled code is behaving the same on the Mac as it did the mainframe. We'll talk next time about creating Mac user interface features in Fortran.

Upon successful compilation, you can immediately run your new codefile by double clicking the **Random Test** .apl file on the desktop. This will cause the MacFortran run time library (`f77.rl`) to be dynamically linked in, as well as the external subprogram `toolbox.sub` the first time it is called. This approach of dynamic runtime linking can be useful if you have a set of commonly used subroutines.

## Linking to Standalone

One of the stated goals at the start of the article was the ability to link the resulting code into a single stand alone clickable application. This can be done with the MacFortran linker. To link the example program double click the linker and execute the following linker commands in response to the > prompt:

```
>f Random Test
>f toolbox
>l f77.rl
>o Random Program
> (carriage return)
```

Be sure that the files **Random Test** .apl, `toolbox.sub`, and `f77.rl` are all on the same disk and that the commands are executed in the above order. The primary restriction is that the runtime library should always be the last file linked. This will generate an approximately 22k application file (2k from **Random Test** .apl, 3k from `toolbox.sub`, and 17k from `f77.rl`, the fortran run time support module

required of any stand alone program).

While linking a stand alone file in this sample program was easy enough, that is not always the case with the MacFortran linker. The linker can get lost sometimes and report no unresolved references, even when some still exist. Notably, this situation seems to arise when there is a call to an external function which references a function which is external or internal in the calling subprogram's parameter list. For example, the following call would not be handled properly:

```
KU2=toolbox(BITXOR,KU1,toolbox(BITAND, LU1*N2TCM,KC))
```

This is not a problem at run time for unlinked external subprograms because the runtime program will dynamically search the disk to resolve any unresolved reference. The problem arises only with programs linked with the linker. This problem is being corrected by Absoft in the soon to be released version 2.1 which will contain a completely rewritten linker. A short term strategy, which may or may not work, is to link to the compiled code to the external procedures, then relink that code again to the external procedures, and then link the run time library. Another notable change in version 2.1 is that the `toolbox.sub` file will shrink from approximately 3k to 500 bytes and the runtime library will stay about the same size.

## Gripes with Lack of Mac-like Features

*The following several paragraphs were added by the Editor based on his efforts to compile and link the Random number program presented in this article.*

[The present linker as well as the other Absoft routines unfortunately show that this product is a port-over job from another computer. The Mac interface is poorly implemented. The dialog boxes are rather simplistic and unprofessional in appearance. The 'about' dialog box looks at first glance



like an error message! The protection scheme is also implemented in a rather shoddy manner. Instead of a simple statement asking you to insert the master disk for verification, the dialog box makes you click a button to eject the disk, insert the master, then click another button to say "OK, I've inserted it!", and finally insert the system disk back in the drive after the master is ejected. A very tedious procedure considering you have to do it every time you boot up. They could have at least ejected the disks for you and detected the disk insertion event rather than making you do all the work. The quality of the buttons in the dialog boxes also looks like a rush job to merely get the Mac interface up and running. The linker is especially bad, in that it doesn't even have a menu bar! Strictly keyboard commands! There is really no excuse for that at this late date in the Mac life cycle. The compiler also operates in a funny way, again showing haste in implementation. To compile a file, you first select the file from the file menu, then open it, (nothing appears to happen at this point) then go to the compile menu to compile whatever it was you opened way back when. Again, this is very roundabout. The expected behavior is to simply invoke the compile command, get a standard file dialog box and click on the file to be compiled, and be off and running. This is the way it should work, but doesn't in version 2.0. We will report if this user interface is cleaned up in 2.1.

### No Error Messages

My final gripe is cryptic error messages. If the disk fills up, you get a XX error code and a disk full of locked intermediate files you can't get rid of without shutting down and rebooting everything or running another application to release the "in use" lock. More meaningful error messages, or better still, alert boxes allowing some remedy would be nice. And the fact that the files created by the compiler are all locked as "in use" after the compiler gave up has got to be a bug. On the positive side, the compiler does have a transfer menu that lists the MDS editor

or MacWrite. Nice. However, they neglected to adjust the companion transfer menu in Edit so the transfer is only one way. Edit doesn't know about the fortran compiler; it simply shows the Apple assembler as grayed out. I tried compiling the assembly source code output to see what would happen and I found out. The compiler doesn't have an escape button to eject from a compiler gone mad condition. So I had to wait while the compiler choked its way through the assembler file, flashing countless error messages at me. A simple cmd-period escape would be nice.

### Assembly Source or Object Code

Something to be aware of is when you select the assembler source code compiler option, no object code is generated! Apparently you either get the object code or the assembly source, but not both together. This caught me by surprise. I couldn't figure out why I wasn't getting any output from the compiler. The output of the compiler (when you *don't* select asm source option) is a 3K code resource file placed on the desktop with a standard application icon. This is deceiving, however. This file is not a true application since it can't run by itself. The Absoft system dynamically links this file to the run time support package when you click on it. If that package is not available you get a beep and a quick trip to the Finder. To get an honest-to-gosh application, also under the standard icon, you have to invoke the linker and execute the link commands as shown previously. Then you get a 23K file that can go anywhere.

True double-clickable applications, fully compiled to 68000 machine code. This is the only available true compiler for the Macintosh outside of the C compilers. As of this writing no other language on the Mac can be compiled to 68000 object code except C. Can the assembly source code be assembled with the MDS system and linker into an application? The answer is YES!

I ran the assembly source code through the MDS assembler and it

worked! The only change was the ".START" label had to be changed to "START" and declared external for the (not so smart) MDS linker. The resultant object code file can be linked by the MDS linker to resource files, custom icons, you name it. The output of the MDS linker is an application, but one that will attempt to open the Fortran run time module. When you double click on it, the Fortran file in question is opened (so it must be available) and the program executes. I then tried to take this MDS Linker output (a code resource file) and link it to the Fortran run time module using the Fortran linker. For some reason, this failed. The Fortran linker would not recognize the program file. In summary, you can compile a Fortran source program to assembler source code or object code; you can link the object code to create a stand alone application or you can assemble the assembly source code with the MDS system and link it with files created under the MDS system but the resulting application will require the Fortran run time module to be present on the disk. There probably is a way to get the Fortran linker to link MDS applications to the run time module, but that will take more investigation. -Ed.]

### Testing the Rand Function

Running the Random Test apl reveals two interesting features about the performance of the random number routine and MacFortran. The test program runs 1000 trials of sample size 60 on random numbers between 1 and 6. The average number of outcomes for a particular value in a sample of 60 rolls should be 10. Every time I have run this test, the average number over 1000 trials has been within 1 or 2 percent of 10 for each value between 1 and 6. This is a crude indication of a fairly reliable random number generator. The second feature to note is the length of time to execute the 60,000 executions of rand and assign the outcomes to the counting bins: 106 seconds for the unlinked program and 92 seconds for a fully linked version. Conversations with Absoft revealed that

the first two loops in a nest are handled with registers in the compiled code. This helps explain the speed of the program, which is very good. [Those of you frustrated with BASIC should take note! -Ed.]

Well, I hope this gives you an initial idea about porting Fortran programs to the Mac. In future columns we will explore adding menus and windows to your programs, standard file procedures, and other Mac software technology. Until then, if you have any suggestions or tips on Fortran on the Mac, please pass the information along.

```
*
* A MacFortran program implementing a random function
* and doing a rudimentary test of its properties. The
* program uses toolbox calls for bit operators and
* tickcount, but does not use toolbox for custom menus
* or windows.
*
* Mark E. McBride
* 211 N. University
* Oxford, OH 45056
* (513) 523-1438
* July 1, 1985
*
PROGRAM Testrandom
*
* Declarations
*
Integer*4 Y,Tick1,Tick2
Integer*4 toolbox,TICKCOUNT ! toolbox variable
Integer*4 A(1001,6)
PARAMETER (TICKCOUNT=362) ! toolbox trap
*
PROGRAM STARTS HERE
*
Y=Rand(0) ! Seed the random number generator
*
Initilize array, MacFortran will not initialize for you
*
DO 500 I=1,1001
DO 500 J=1,6
500 A(I,J)=0
*
* Get a system tick count using the toolbox routine
*
Tick1=toolbox(TICKCOUNT)
write(9,*) ' System tick count at start is ',Tick1
write(9,*) ' more coming...'
write(9,*)
*
* Do one thousand trials each of sample size 60
*
DO 1000 I=1,1000
DO 1000 J=1,60
Y=int(Rand(1)*6+1)
1000 A(I,Y)=A(I,Y)+1
*
* Get a second system tick count
*
Tick2=toolbox(TICKCOUNT)
*
* Calculate total number of responses for each value
```

KU1=toolbox(BITXOR,KU1,LU1/N2TM)

```
MOVEM.L D5/D6/D7/A0/A2/A3/A4/A5,-(A7)
SUBQ.W #6,A7
MOVEQ #89,D0
MOVE.L D0,-(A5)
MOVE.L A5,-(A7)
PEA 8(A3)
MOVE.L 8(A3),D0
MOVE.L 24(A3),D1
JSR 68(A4)
MOVE.L D0,-(A5)
MOVE.L A5,-(A7)
MOVEQ #3,D0
MOVE.L #2137475944,D1
JSR 4(A4)
ADDA.W #18,A7
MOVEM.L (A7)+,D5/D6/D7/A0/A2/A3/A4/A5
MOVE.L D0,8(A3)
```

**Fig. 2 Sample Line of Fortran In Assembly Source Code Format from the Compiler Output**

```

*
 DO 1500 I=1,1000
 DO 1500 J=1,6
1500 A(1001,J)=A(1001,J)+A(I,J)
*
* Write out results to MacFortran generated output window
*
 Write(9,*) ' The average # of responses in sample
+ of size 60 after 1000 trials is'
 Write(9,*)
 DO 2500 J=1,6 ! write average response rate
2500 Write(9,101) J,float(A(1001,J))/1000
 101 Format(/1x,' For die value ',I2,' the average
+ number of responses was ',f8.5)
 write(9,*)
 write(9,*) ' System tickcount at end is ',Tick2
 write(9,*)
 write(9,*) ' Sampling took ',float((Tick2-Tick1)/60),
+ ' seconds'
 write(9,*)
 PAUSE 'Press Return to quit' ! hold the screen
 STOP
 END
*
* Include the code for the random number generator
*
 include Rand.for
*
* end of source code
*

FUNCTION Rand(IX)
*
* This random number generator is a variation on Tausworthe
* generator described in "Solution of Statistical Distribution
* Problems" by H. O. Hartley in Statistical Methods for Digital
* Computers Vol III, edited by Enslein, Ralston, and Wilf
* (John Wiley & Sons 1977). The only modification of consequence
* is the ability to reseed the generator with the system
* tickcount.
*
* The function returns a real value between 0 and 1.
* Rand(0) will reseed the random sequence using system tickcount.
* Rand(1) will use previous calls values as seeds for next number
* in the sequence. Always using Rand(1) will generate a specific
* random sequence based upon starting values internal to
* the function.
*
 Real*4 Factor,KU1,KU2

 Integer*4 Toolbx,BITXOR,TICKCOUNT ! toolbox definitions
 Integer*4 KU3,LU1,KC,N2TM,N2TCM

 Equivalence (KU2,KU3),(KU1,LU1)

 PARAMETER (BITXOR=89) ! toolbox definition
 PARAMETER (TICKCOUNT=362) ! toolbox definition

* Note that the parameter statement assigns a value to
* the indicated constant label. In this case, it is the index
* into a table of trap addresses contained in TOOLBX.SUB which

```

- \* converts the call to a trap call. A list of these assignments
- \* is given in the file TOOLBOX.PAR.
- \* You must explicitly tell MacFortran to save the values of
- \* local variables across successive calls of the subroutine.

```

SAVE
Data Factor/0.4656613e-9/
Data N2TCM/z'00040000'/
Data KU1,KC,N2TM/z'40000003',z'7fffffff',z'00002000'/

```

- \* KU1 enters with U(I) uniform random variable
- \* KU2 leaves with U(I+1) uniform random variable
- \* KCU1 complement of KU1
- \* KCU2 complement of KU2
- \* KC complementing constant
- \* LXXX logical equivalences of above KXXX
- \* N2TM 2\*\*M where M is shifting factor
- \* N2TCM 2\*\*P-M where P is word size
- \* Factor float value of 2\*\*P
- \* XOR is the exclusive or operator

```

if (IX=0) KU1=toolbx(TICKCOUNT)
KU1=toolbx(BITXOR,KU1,LU1/N2TM)
KU2=toolbx(BITXOR,KU1,(LU1*N2TCM).and.KC)
Rand=FLOAT(KU3)*Factor
KU1=KU2
RETURN
END

```



# *Fortran's World*

## *Fractals in Fortran 2.1*



*Chuck Bouldin*  
*MacTutor Vol. 1 No. 11*

### **A Look at MacFortran 2.1**

Last month's issue contained a first look at Macfortran, an implementation of FORTRAN on the Macintosh. This article is intended to continue that discussion, and specifically to look at the new features in the 2.1 release of Macfortran and begin to explore the Macfortran interface to the Toolbox routines. This article will describe the new Toolbox capabilities in the 2.1 release, the extensions and improvements to FORTRAN in 2.1 and give a couple of short examples which begin to show how Macfortran interacts with the Macintosh environment. It will discuss the strengths (and weaknesses) of this implementation of FORTRAN, and show what this language can accomplish on the Macintosh.

### **Why Fortran?**

One of the first things to determine about any computer language is the class of problems for which it is best suited. There are lots of languages now available for the Macintosh; we need to determine where FORTRAN fits into the hierarchy of tools available for programming the Mac. Please note that this is **not** intended to continue the meaningless debate about the 'best' computer language, rather this is intended to be a (reasonably) realistic assessment of FORTRAN as a tool on the Mac.

First of all, there exists a very clear definition of the language. The American National Standards Institute (ANSI) has published a document which completely defines the FORTRAN 77 language. Any implementation of the language must meet the standards defined by ANSI, or it really isn't FORTRAN. In the past, the main problem with microcomputer implementations of FORTRAN has been that they were incomplete, including only a part of the ANSI standard. MacFortran is a refreshing break from the past; it implements the full ANSI definition of the language. Since the language is so clearly defined, this makes for a remarkable degree of software portability. Code written in FORTRAN on any other machine, micro or mainframe, ports easily to the Macintosh. Obviously, Macintosh extensions such as Toolbox calls will not be of much use on other machines, but the other parts of the code are standard. Details on the Toolbox interface and porting FORTRAN from other machines to the Mac are given later.

Second, MacFortran is the only fully compiled language for the Macintosh other than C. This means that when an application must run **fast**, then the high level language possibilities are C or FORTRAN. In some ways FORTRAN represents a natural upgrade from Basic when speed is a

concern, since Basic was first derived as a subset of FORTRAN. Basic programmers who find they need faster running programs will probably find it easier to convert to FORTRAN than to learn C.

Third, if you crunch numbers, this is your language. MacFortran does not use SANE, the 80 bit software floating point built into the Mac. MacFortran has its own floating point; 32 bit single precision and 64 bit double precision. This floating point is fast, easily the fastest available on the Macintosh. Specific benchmarks are given below.

The language does have limitations, which show up mostly in the ancient lineage of FORTRAN. The most glaring limitation is the inability to create data types which correspond to Pascal records or C structures. The closest that MacFortran comes to this is simple arrays. This means that when dealing with complex data structures, the programmer, rather than the compiler, is forced to do some of the bookkeeping about where various data elements are located.

### **Toolbox Support**

Of course the crucial element of a language for the Macintosh is how well it can interface to the Toolbox routines provided in the Mac ROMS. MacFortran provides direct access to almost all of the Toolbox routines, including Quickdraw, access to serial ports, Window and Menu creation, the font manager, the standard file package, Text edit, use of Desk accessories from inside FORTRAN programs, and more. Missing so far are access to the control and memory manager. Although not all of the Toolbox is directly supported from FORTRAN, MacFortran allows calls to assembly language subroutines which can obviously access any routine.

MacFortran calls Macintosh routines through a single subroutine, TOOLBX. The first parameter passed to this subroutine is an upper case character string which is the name of the Mac routine being called. The parameters which follow are the arguments that the Mac routine needs. TOOLBX is a 'glue' routine which takes care of putting the arguments onto the stack or into the cpu registers.

There is one main complication in using MacFortran to access Mac Toolbox routines. As mentioned earlier, FORTRAN does not support data types such as Pascal records or C structures. Since such data structures are used by many of the Toolbox routines, how can these routines be supported from MacFortran? The answer is simple, the data structures are just mapped onto a MacFortran array, and the array is then passed to TOOLBX, which handles the mapping from array elements onto stack locations or 68000 registers. With this type of approach, MacFortran can handle all of the Mac

Toolbox with equal facility.

### Using MacFortran

The MacFortran system comes with compiler, linker, librarian, and source code debugger. Apple's Edit and Rmaker programs are also included. The compiler and debugger have full Mac-style interfaces with pull-down menus and the usual dialogue boxes for file selection and compiler options. Suprisingly, the linker and librarian retain Unix type interfaces.

The compiler is organized into 3 overlays: (1) parsing and preliminary reduction of tokens into object code, (2) resolution of backwards jumps and some final code generation (3) setting up pointers for a runtime linkage. The overall compilation speed for large programs is 500-1000 lines/minute. Due to disc overhead swapping the compiler overlays, small programs (under 100 lines) will seem much slower, perhaps 50-200 lines/minute.

As mentioned above, the third compiler pass actually sets up pointers for a runtime link. In a program which does not call previously compiled subroutines this just sets a pointer to load the MacFortran runtime library. At runtime, the library and any other needed subroutines, will be autoloaded when called. At runtime, first the default disc and then the internal disc are searched for needed subroutines. During development, this saves having to crank up the linker every time a change is made in the code. On the down side, subroutines are not locked in memory after they are called once; they are reloaded at each call. Clearly, you don't want runtime linking for subroutines which are called more than a few times. Also, the search path can lead to some confusion. If an application program is executed from the internal drive and the runtime library is in the external drive, the runtime library is not found, since the internal drive is now also the default drive. The internal drive is first searched as the internal drive and then as the default drive, missing the runtime library. [Isn't this a bug? -Ed.]

Easily the most impressive component of the package is the source code Debugger. This utility brings up a window of the MacFortran source code being debugged, a window to display variables and a 5 line MacFortran output window. The source code can be single stepped, breakpoints can be set and changed, and variables can be continuously monitored. This is almost like having a FORTRAN interpreter! Getting code working is very greatly simplified by this debugger.

The linker and librarian are certainly functional, but do not have a Macintosh type interface. The worst part of this is having to remember file names. On the plus side, the linker supports creation of 'scripts', files which contain canned linker 'programs'. This simplifies linking since many of the same libraries and subroutines are linked to different programs. According to Absoft there are two reasons for the non-Mac interface: (1) to provide the large table space that the linker needs and still fit into a 128K machine it was necessary to use a primitive interface, and (2) the interface remained unchanged in 2.1 since the programming effort was put into

changing the linker to support linking on an entry point, rather than a filename basis. In 2.0, the ENTRY statement in FORTRAN was effectively disenfranchised since linking was done on only a filename basis. I still don't like the user interface, but the functionality of supporting the ENTRY statement is certainly more important. [In other words, the linker isn't finished yet! -Ed.]

The linker also makes no discrimination about what parts of the runtime library are actually used by a program; the entire library is linked in regardless. This means that the smallest MacFortran program will be about 18K, a real detriment to writing desk accessories in this language. On the other hand, several programs can be completely linked except for the runtime library and share a single copy of the runtime library through runtime linking. Also, applications that do anything more involved than "hello, world" are likely to bring in large fractions of the Runtime library anyway. Runtime linking is certainly not required. The linker and librarian manage pre-compiled object code and standalone applications are easily created.

### Portability Considerations

As mentioned earlier, one of the main attractions of FORTRAN is the portability of the language. I have ported several large applications down from a VAX 11/780 and have learned some of the pitfalls to watch out for when porting code down to the Mac. Since the use of existing code, particularly mathematical software, is a major incentive for using Macfortran, it is worth describing the portability considerations in detail. All of the non-portable code that I have downloaded to the Mac has turned out to be due to the use of non-standard features in VAX FORTRAN.

First, it is essential to use the SAVE statement in order to preserve variables between calls to a subroutine. This was mentioned in last month's column, but the impression was given that this is a quirk of Macfortran. It is not. It is standard FORTRAN, although VAX and most mainframes do not require it. On a small machine such as the Macintosh, the use of the SAVE statement allows you to write larger programs by disposing of unneeded variable space. The main caution is that COMMON blocks must be SAVE-d between calls to a subroutine. This is counterintuitive, but it is standard FORTRAN.

Second, DATA statements must follow all declarations (REAL, COMPLEX, etc.). Again, this is standard FORTRAN, but is relaxed on most large machines. The main place to look out for this is when using INCLUDE statements to enter the COMMON variables in a group of subroutines that share COMMON blocks.

Third, watch out for READ/WRITE statements that use system dependent logical units. The way to avoid this problem is to use PARAMETER statements to assign logical units. For instance, to avoid the problems described last month with units 5 and 6 in Macfortran I use on the Macintosh:

LU6=9

```
LU5=9
WRITE(LU6,*)
READ(LU5,*)
```

On the VAX I use the same code, but set:

```
LU5=5
LU6=6
```

It is also possible to use the compiler option which maps READ\* and PRINT\* statements to unit 9 to resolve all the problems associated with list directed READ and WRITE statements.

### Extensions and Enhancements In 2.1

The major news for the 2.1 revision is that the compiler has been licensed to Microsoft. This should mean that there will now develop a much larger user community for this language. In evaluating the 2.1 revision I have been working from a (slightly) modified version of the 2.0 documentation. According to Absoft, the new manual has been completely rewritten by Microsoft, and is much more Mac specific than the old documentation. The manual is 400 pages, organized into a Users guide and a Reference manual. It also reflects an orientation toward a less sophisticated user with less prior knowledge of FORTRAN.

In addition to Macintosh specific enhancements, there are some extensions in the Absoft FORTRAN which are very nice. Probably the most powerful is the addition of recursion to the language. Some programs are very difficult to write without recursion and are very natural when recursion is used. As an example of recursion, and to illustrate the drawing speed of Macfortran a short example which draws fractals is given below.

Other extensions include a CASE statement for multi-way branches, DO loops that are terminated with a REPEAT statement, and a generalized looping statement, WHILE(condition)...REPEAT. The WHILE statement is especially useful, since it gives much more flexible looping to FORTRAN; one no longer has to map all loops onto arithmetic progressions.

The compiler still operates in the same way as before. The user selects a file from the standard file box and then chooses among options such as Compile, Compile and Execute, Compile and Debug, etc. A nice human engineering change is that the compiler options can now be saved so that they do not need to be reset after every compile. The user still has to explicitly choose an option before anything happens. I do not find this flexibility confining.

The standard file package, Text edit utilities, and Desk accessory support (Systemclick and Systemtask) are now accessible from Fortran. The glaring things still missing are access to the Control and memory managers. By using the compile to assembly source to observe what code is generated for, say, the Window Manager it should be possible to figure out what to do for the Control Manager. It is aggravating to

have to do this; hopefully we will soon get full documentation on all the Toolbox.

There is now a compiler option to map lower case onto upper case. This is very useful since only upper case characters are recognized by an ANSI standard compiler. Differences between VAX and Macfortran about character case have caused problems porting code. This option switch solves those problems.

There are several new demo programs, including a Fortran version of the Edit program from Inside Macintosh. On the subject of Edit, it is not Absoft's fault that the MDS Editor contains hardwired references to its transfer points. This is not even a resource that can be changed with the resource editor! Instead, references to EXEC, etc. are embedded in the code in the MDS Editor. The simple (but not elegant) solution is to rename Fortran as EXEC on the desktop. This allows transfers back and forth between Macfortran and Edit.

There is now access to the A5 register and Quickdraw globals via a call to GETGLOBALS.

### Performance

As mentioned earlier, one reason for using this language is speed. The code is fully translated into 68000 object code for fast execution. For instance, the Sieve of Eratosthenes runs in 6.8 seconds, about as fast as Mac C compilers (without register variables). It is harder to find such universal benchmarks for floating point, but MacFortran executes single precision floating point 5-10 times faster than Mac languages that use the 80 bit SANE package. Some of the Mac C compilers also have their own floating point packages, but MacFortran is 2 times faster than the fastest of these.

To give some idea of floating point speed I ran two benchmarks that have had some runs on other machines (mostly IBM and clones). I ran Jerry Pournelle's benchmark which fills and multiplies two 20x20 matrices, with an execution time of 2.9 seconds. This is very fast, since an 8 Mhz 8088 with 8087 runs the same benchmark in 3.9 seconds using Pascal MT+. In addition, I ran the benchmark 'float' from the Aug, 85 issue of Byte magazine, page 133. Here, an IBM XT with 8087 recorded a time of 11.7 seconds using Microsoft Fortran and an IBM AT with an 80287 and Digital Research Fortran came in at 17.7 seconds. Using MacFortran, the Macintosh comes in at 7.7 seconds. When compared to a VAX 11/780 for large application programs such as matrix inversion, Fourier transforms, and optical ray tracing Macfortran delivers about 4-7% of the 780 performance. Of course, what a user cares about is how long he waits for the results. On a moderately loaded VAX I find that the difference in execution time drops to a factor of 2-5 between the Mac and the VAX.

I won't belabor the comparisons any more since the conclusion is clear; MacFortran is very, very fast for floating point operations. For a reasonable mix of floating and fixed point operations, MacFortran is at least as fast as any other language on the Mac. It is also faster than the main microcomputer competitor even when compared with

**hardware floating point.** For crunching numbers, this language is unbeatable.

There are some caveats. The above benchmarks compare the default precision of each language. In MacFortran this is 32 bit single precision, providing only 5-6 decimal digits of accuracy, so is it 'fair' to compare this with the 80 bit software floating point of SANE or the 80 bit hardware floating point of the 8087? Yes, it is. Languages which do all floating point as 80 bit calculations give the user no choice, frequently providing more precision than is needed. In MacFortran you choose; if you need more accuracy, use double precision (64 bit), while if speed is the main concern, stay with single precision. Usually single precision is fine.

### A Pair of Examples

The examples below are intended to illustrate a small part of the Macfortran interface to the Mac Toolbox routines. These show about the simplest level at which a FORTRAN program can interface with the Macintosh. The 'user interface' in these examples is not via Menus and dialogue boxes, but these routines do show off the graphics capability available from Macfortran and illustrate the use of recursion for drawing fractals (see Aug, 85 Macworld for the same example in MacPascal).

```
program ranplt
```

```
*
```

- \* This illustrates the use of some Quickdraw routines for plotting on the screen and erasing areas of the screen. This routine is intended to show the drawing speed on the screen and to illustrate the use of the internal random number generator, TOOLBX(RANDOM). Since the coordinate arrays are filled before plotting starts, this routine gives a measure of the raw plotting speed from inside Macfortran. Toolbox routines used are:

```
* ERASERECT
* RANDOM
* LINE
* MOVETO
```

```
* Storage for the starting position
 INTEGER XSTART
 INTEGER YSTART
```

```
* For random coordinates
 INTEGER*4 IXRAND(3000), IYRAND(3000)
 INTEGER*4 TOOLBX
```

```
* Declarations for calculating elapsed time for plot
 INTEGER TIME1, TIME2
```

```
* Include the Toolbox Call Parameter file.
 INCLUDE TOOLBX.PAR
```

```
* Rectangle stuff for erasing the screen
 INTEGER*2 BIGRECT(4), SMLRECT(4)
 DATA BIGRECT /0,0,512,512/
```

```
DATA SMLRECT /0,0,60,100/
```

```
* Stuff default values for NSTEPS and ISCALE
 NSTEPS = 1000
 ISCALE = 6
```

```
* Set up number of steps and stepsize
 PRINT*, 'Number of steps ? (3000 maximum, 1000 default)'
 READ*, NSTEPS
 PRINT*, 'Approximate step size in pixels ? (5 default)'
 READ*, ISCALE
 ISCALE = 32000 / ISCALE
```

```
* Clear the screen
 CALL TOOLBX(ERASERECT,BIGRECT)
```

```
* Get the starting time (in ticks)
* Note that this is a system global!
 TIME1=LONG(362)
```

```
* Stuff the random coordinate arrays
* TOOLBX(RANDOM) returns a random integer between
* -32768 and 32767, inclusive
 DO (I=1,NSTEPS)
 IXRAND(I) = TOOLBX(RANDOM) / ISCALE
 IYRAND(I) = TOOLBX(RANDOM) / ISCALE
 REPEAT
```

```
* Move to center of screen
 XSTART = (BIGRECT(1) + BIGRECT(3)) / 2
 YSTART = (BIGRECT(2) + BIGRECT(4)) / 2
 CALL TOOLBX (MOVETO,XSTART,YSTART)
```

```
* Draw the next line.
 DO (I=1, NSTEPS)
 CALL TOOLBX (LINE,IXRAND(I),IYRAND(I))
 REPEAT
```

```
* Get the ending time and print deltat
 CALL TOOLBX(ERASERECT,SMLRECT)
 CALL TOOLBX(MOVETO,10,10)
```

```
* Ending time in ticks
 TIME2 = LONG(362)
 DELTAT = (TIME2 - TIME1) / 60.
 PRINT*, 'ELAPSED TIME = ',DELTAT
 PAUSE
 END
```

```
program snowflake
```

```
* Draws a fractal pattern using Toolbox calls and recursion.
* Calls
* ERASERECT
* MOVETO
* LINETO
*
```

```
include quickdraw.inc
integer x1,y1,x2,y2,n
integer xorig,yorig,scaling
integer*2 rect(4)
xorig = 250
```



```

yorig = 230
scaling = 100
rect(1)=0; rect(2)=0; rect(3)=340; rect(4)=510
do
 call toolbox(ERASERECT,rect)
 call toolbox(MOVETO,3,12)
 type "order= (0 to quit)"; accept n
 if(order.eq.0) then
 go to 20
 endif
 call toolbox(ERASERECT,rect)
 x1 = xorig - scaling
 y1 = yorig
 x2 = xorig + scaling
 y2 = yorig
 call snowr(x1,y1,x2,y2,n)
 pause
 repeat
20 continue
end

subroutine snowr(x1,y1,x12,y12,n)
include quickdraw.inc
integer x1,y1,x12,y12,n
integer xdiff,ydiff,x(2:12),y(2:12)
s = 0.5573
x(12) = x12
y(12) = y12
xdiff = x12 - x1
ydiff = y12 - y1
x(10) = x1 + xdiff/3
y(10) = y1 + ydiff/3
x(11) = x1 + xdiff*2/3
y(11) = y1 + ydiff*2/3
x(3) = x(10) + ydiff*s

```

```

y(3) = y(10) - xdiff*s
x(9) = x(10) + ydiff*s/3.0
y(9) = y(10) - xdiff*s/3.0
x(8) = x(10) + ydiff*s*2.0/3.0
y(8) = y(10) - xdiff*s*2.0/3.0
x(2) = (x1+x(3))/2
y(2) = (y1+y(3))/2
x(4) = x(11) + ydiff*s
y(4) = y(11) - xdiff*s
x(6) = x(11) + ydiff*s*2.0/3.0
y(6) = y(11) - xdiff*s*2.0/3.0
x(5) = (x12+x(4))/2
y(5) = (y12+y(4))/2
x(7) = (x(8)+x(4))/2
y(7) = (y(8)+y(4))/2
if (n=1) then
 call toolbox(MOVETO,x1,y1)
 do (i=2,11)
 call toolbox(LINETO,x(i),y(i))
 repeat
 call toolbox(LINETO,x12,y12)
 else
 call snowr(x(2),y(2),x1,y1,n-1)
 call snowr(x(2),y(2),x(3),y(3),n-1)
 call snowr(x(3),y(3),x(4),y(4),n-1)
 call snowr(x(4),y(4),x(5),y(5),n-1)
 call snowr(x(5),y(5),x(6),y(6),n-1)
 call snowr(x(7),y(7),x(6),y(6),n-1)
 call snowr(x(7),y(7),x(8),y(8),n-1)
 call snowr(x(9),y(9),x(8),y(8),n-1)
 call snowr(x(9),y(9),x(10),y(10),n-1)
 call snowr(x(11),y(11),x(10),y(10),n-1)
 call snowr(x(11),y(11),x(12),y(12),n-1)
 end if
end

```



# Fortran's World

## Shell Application for MacFortran 1.2



Mark E. McBride  
Oxford, OH  
MacTutor Vol. 1 No. 12

This month's column deals with designing an application shell to be used with ported Fortran programs. While the basics of an application shell have been documented in other MacTutor articles, I felt a shell in Fortran would give users a good starting point for integrating their ported programs.

### Porting to a Shell

Most mainframe Fortran programs can be viewed (at least for our purposes) as a straightline or sequential process. The user starts the program, it executes (probably not in a pure straightline fashion), and it stops at one or more predefined points in the code. Rewriting this straightline code into a Mac user interface will probably (always?) involve massive restructuring of the code. Given that the third of the stated goals in the previous month's column was minimizing conversion effort, complete restructuring is likely to be undesirable in a large number of situations. This presents the basic dilemma: the user wants to port Fortran programs from the mainframes with a minimal amount of conversion effort, but the user does want to include a Mac user interface.

One solution to this dilemma is to treat the ported program as a subroutine, to be executed upon request from a menu choice. This lets the user have a covering application shell which has the standard Mac features available until execution of the ported program begins. Just before starting execution of the ported program, the Mac features can be completely or selectively disabled. Upon completion of the ported program, the user is returned to the application shell. This design gives the user the ability to rerun the ported program several times without exiting to the desktop.

Of possibly greater benefit to the user from this design is that it provides a starting point for integrating the ported Fortran program. If no further integration is desired, none needs to be done. However, in the application shell the user has the basic structure in place for integrating those Mac features desired to the ported program. This could even

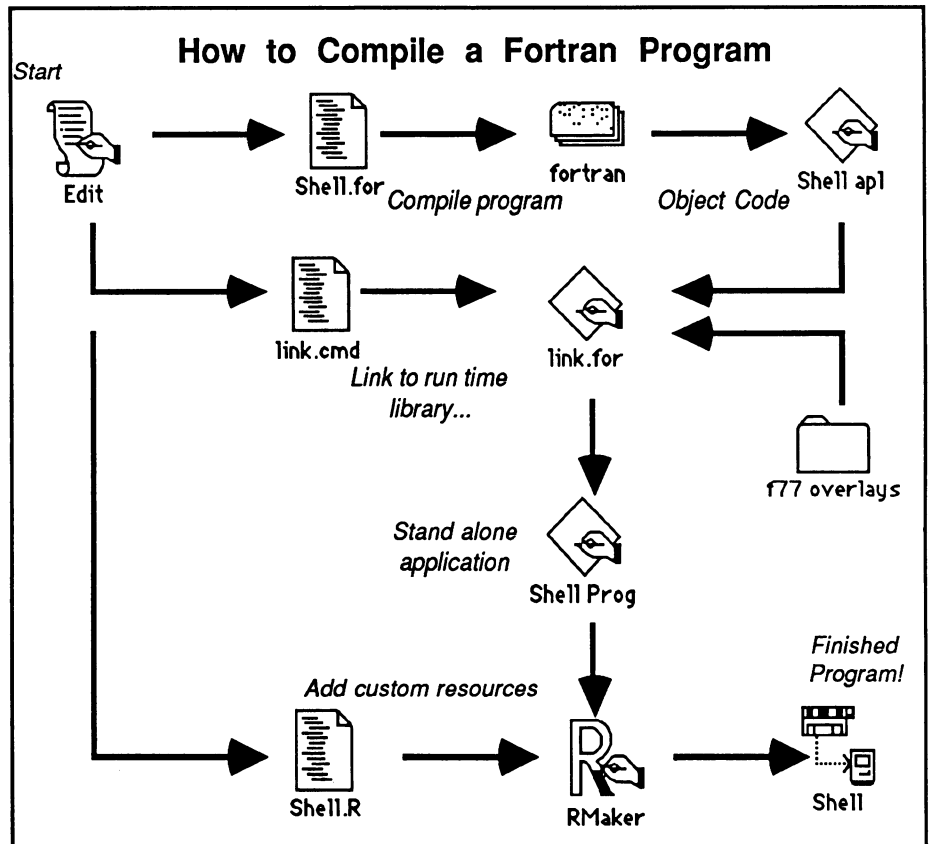


Fig. 1 Fortran Compilation

include importing information via desk accessories and/or cut, copy, and paste. Users have the freedom to integrate as much or as little as they like.

### The Shell Program

The application shell given in Listing 1 has parentage from several other programs. The initial basic design came from the Demo.for program included with MacFortran. However, the program goes well beyond that demo. Design ideas also came both from a translation to Fortran of the sample edit program on the developers disks (graciously provided by Absoft) and from previous MacTutor articles. I find that the assembly shell by Dave Smith (Vol 1 #'s 3 and 4) and that the C Workshop series by Bob Denny (Vol 1, #'s 2 thru 7) very useful for design assistance. Reviewing these articles yields many insights into the program design. In fact, I wrote the shell mostly from review of the aforementioned articles and program listings, with very little direct reference to

*Inside Macintosh*. The program is written using MacFortran version 2.1, now being distributed by Microsoft, and in particular, uses the include files from the 2.1 version.

The shell program makes heavy use of the toolbox using the `toolbx.sub` subroutine calls. Toolbox procedures that return a value are accessed by the `toolbx` function call. For example: `mywindow = toolbx (FRONTWINDOW)` would return a pointer (integer\*4) to the front window on the screen to the variable `mywindow`. Toolbox procedures that do not return variables are accessed as a `toolbx` subroutine call. For example: `call toolbx (DRAWMENUBAR)` executes the toolbox routine that draws the menu bar given the current menu list.

Given that MacFortran does not type or range check values passed to the `toolbx` routine, it is a good idea to compile your programs using an *implicit none* statement in the main and all subprograms. This forces the compiler to report any undeclared names. This is particularly important for the names for the `toolbx` calls.

[Access to information contained in toolbox parameters is done by defining appropriate length data constructs, then using equivalence statements to access those elements needed. In converting this program from 2.0 to the Microsoft version 2.1, a problem arose regarding the use of the include files. The include file for the Event manager contains the definitions to translate the Pascal type event record structure into a fortran array. Equivalence statements then equate labels such as `what`, `where` or `modify` to the appropriate array item. In most of the include files, only the dimension statements are given; the equivalence statements should be in your main program. However, in the Event include file, equivalence statements are given for the event record array to the typical MDS field names mentioned above. Since this include file must be present in all your subroutines, a subtle error can occur where an equivalenced variable (from the Event include file) is also referenced as the dummy argument in a subroutine parameter list. In the program listing in figure 1, I had this problem. Rather than change the include file, I used another dummy variable, `mouse`, to replace the `where(1)` field in the parameter list for the `Menu` subroutine, thus eliminating the conflict with the fact that the `where(1)` variable was equivalenced to `Eventrecord(6)` in the Event include file. A cleaner way to solve this problem, and what you should do is to remove all the equivalence statements from the Event include file and any other include files, leaving those files only for variable definitions. Instead place the equivalence statements in your main program and that way, they won't conflict with dummy variables in external subroutine calls. -Ed.]

Examination of Listing 1 reveals that the program consists of three main parts. First, the main program initializes the menu bar and window using resources created by the `Rmaker` utility. Listing 2 gives the input file for `Rmaker` to create these resources. `Rmaker` is extremely picky about syntax. For example, a blank line means a blank line; there can be no characters other than a carriage return in that line. Use the `show invisibles` option of the editor to locate any extraneous characters. The resources are retrieved using toolbox calls (e.g. `GETMENU`).

During the initialization of Mac features, the default I/O window provided by the MacFortran run time library is removed by getting a handle to it using `window = toolbx (FRONTWINDOW)`. The window is then closed by `call toolbx (CLOSEWINDOW, window)`. This is all well and fine, except when you try to run the debugger on a program that removes the default window in this manner. When you are using the debugger, the call to `FRONTWINDOW` will return a handle to the statement list window since it is the active front window on the screen, rather than the default I/O window. I typically do not close the default I/O window until the program is fully debugged.

A further note about the debugger is in order. Given the resource file shown in Listing 2, the debugger will not work, because the file type and creator have been changed to allow a custom icon. The strategy for debugging that I follow is to have two resource files: the full file that includes icon resource information and bundling information, and another shorter resource file, that is compatible with the debugger (ie does not have icon, bundle or `fref` resources). This second resource file adds resources to an existing file by using

`!Shell apl`

at the start of the file to identify the existing unlinked compiled code file. The only resource types added are the menu, window, and dialog definitions. This allows a fairly smooth process for debugging. The source is edited, then compiled with the symbol table option for debugging on. From the compiler transfer menu, I use the select application option to transfer to `Rmaker` to add the resources. Then I can run the interactive debugger on the compiled code with resources attached.

After initializing the Mac features, the program enters the event loop. This loop detects when an event has occurred (`GETNEXTEVENT`), processes the event, and then returns for the next event in the queue. The loop decodes the event using the select case statement with the "what" portion of the event record. The mouse down event is of primary importance because it locates where the mouse button was clicked using `FINDWINDOW`. The mouse down event is decoded using if-then-else constructs. Menu events are passed to the subroutine "menus" for decoding. This subroutine decodes the menu item chosen using select case constructs.

The only portion of the program design given in Listing 1 that caused serious exploration of *Inside Macintosh* and the MacFortran manual for an answer was the modal dialog box in the About menu choice. Modal dialogs force the user to respond to the dialog by "locking" out all other interfaces (menus, etc.). Calling a modal dialog defined in the resource fork would be done in MacFortran by:

```
dlg=toolbx(GETNEWDIALOG,dlgid,dstorage,behind)
call toolbx(SETPORT,dlg)
call toolbx(MODALDIALOG,userfilt, itemh)
call toolbx(DISPOSEDIALOG,dlg)
```

where `dlg` is a pointer to the dialog structure, `dstorage` is set to 0 to allocate the storage on the heap, `behind` is set -1 to bring the dialog to the front of the windows on the screen, `itemh` is the item number clicked on by the user, and `userfilt` is a pointer to the starting address of the user supplied filter function. The user controls the events on the screen during the modal dialog with the filter function (see Bob Denny's C Workshop Volume 1, # 7 for an example).

### No Filter Functions In MacFortran

The problem with implementing this structure in MacFortran is that there currently is no easy or reliable way to get the handle to the user filter function, `userfilt`. Absoft states that they are working on a fix for this problem. However, in the meantime, users cannot write their own custom modal dialog boxes. I will report any solution as soon I discover one. Note that this problem does not exist for modeless dialog boxes. For an example of a modeless dialog used for the About menu option, see Dave Smith's Assembly Lab (Vol. 1, #4).

Instead of using a modeless dialog for the About menu, I elected to stay with a modal dialog, but use a simple control button from a dialog resource to exit the dialog box.

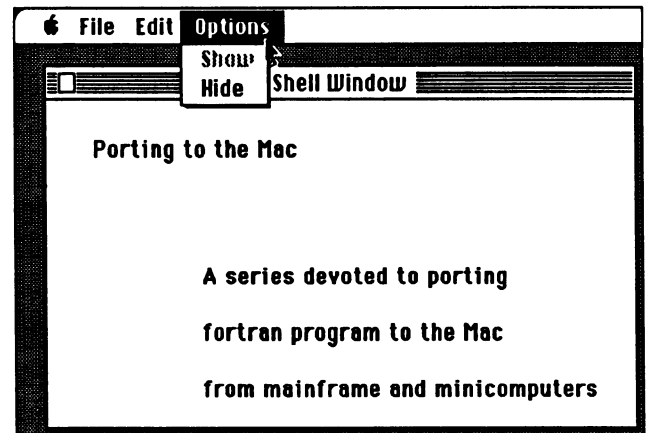
### Creating the Standalone Application

Compiling and linking the resources should proceed in the following manner. First compile the application source code using the MacFortran compiler. Next, if you want to create standalone code, use the MacFortran linker to link the output of the compiler with `toolbox.sub` and `f77.rl`, the run time support package. After linking, then compile your resource file with `Rmaker`, and use an include statement in your resource file to move in the linked fortran code file. If you compile and link the resources using `Rmaker` before linking the runtime package, the MacFortran linker will get lost. So compile, then link, then run `Rmaker`! The result will be a stand alone application file with it's own icon as shown in figure 1.

The required MacFortran linker commands to link the application code to the external subroutines and run time package is:

```
>f shell
>f toolbox
>l f77.rl
>o Shell Prog
>g
```

The `toolbox` subroutine is the external subroutine that handles the toolbox interfacing, while the `f77.rl` file is the library of run time support routines necessary for stand alone operation. Remember to have the linker and all the files on the same disk (I have found the 2.0a linker to be unreliable in getting the run time library from another disk).



### Concluding Remarks

The discussion of the construction of the shell program has been deliberately brief, since most of the programming ideas have been explored in other languages in previous MacTutors. Listing 1 is heavily commented to assist in your analysis of the program. The output of the program is shown in figure 2.

[One area of research that the MacTutor staff is engaged in is the compatibility between the Fortran linker and the MDS linker. As we reported last month, you can compile your fortran routine to assembler source code with the Fortran compiler, and then assemble the source code with the MDS assembler. The only change is that the `START` label must be declared `XREF` and the period prefix removed from the `asm` file. The MDS assembler will then assemble this source and create a `.REL` file that can be linked with other MDS files and resources into an application. But the resulting assembly source code contains an include file `init.asm` that causes the Fortran run time package to be opened, and hence that package must be available when the program is run. The solution to this lies in either getting the MDS linker to link in the `toolbox.sub` and `f77.rl` files to the MDS `.REL` files, or, getting the Fortran linker to open MDS application files. So far, both attempts have failed. The MDS linker does not recognize the two required Fortran routines, and the Fortran linker does not recognize MDS application files. What is needed is further study of the Fortran compiler output file format with the `.REL` file format of the MDS system. Between these two files, a method for compatibility can be found. The motivation for this research is that the new Consulair smart linker is downward compatible with the MDS linker, and hence could be used as a substitute for the Fortran linker, allowing one smart linker to link MDS, C and Fortran files together. We invite reader participation in this effort. -Ed.]

```

*
* The following program demonstrates the MacFortran
* interface to the Macintosh tool box by creating an
* application 'shell'. It uses Quickdraw, the Menu Manager,
* the Window Manager, and the Event Manager. The program
* is originally based on the demo program provided by Absoft
* with the MacFortran package. This program uses resource
* files to load the menu and the window.

```

```

* Mark E. McBride 8/15/85

```

```

program Shell

```

```

implicit none ! helps keep us out of trouble

```

```

include desk.inc
include dialog.inc
include event.inc
include menu.inc
include quickdraw.inc
include misc.inc
include textedit.inc
include window.inc

```

```

* event structures defined in event.inc

```

```

* menu definitions

```

```

integer*4 menuhandle ! pointer to menu

```

```

* window structures

```

```

integer*4 d_window ! pointer to the Shell window
integer*1 d_record(154) ! shell window record

```

```

* general variables

```

```

integer*4 toolbox ! the tool box interface
integer*4 window ! general purpose pointer
integer*4 size,w,h ! for growing documents
integer*2 rect(4) ! rectangle coordinates
integer mouseloc ! mouse location from FINDWINDOW
integer*4 hte,trect(4) ! text edit handle and rectangle
integer*2 mouse(2) ! where() field of eventrecord()

```

```

* Flush the event manager before calling

```

```

eventmask = -1

```

```

* Close MacFortran I/O window

```

```

window=toolbox(FRONTWINDOW)
call toolbox(CLOSEWINDOW>window)

```

```

* Call Text Edit initialization. This is the only package you
* need to initialize. MacFortran handles all the others.

```

```

call toolbox(TEINIT)

```

```

* Build the menu from the resource file

```

```

menuhandle=toolbox(GETMENU,29) ! get 1st menu
call toolbox(INSERTMENU,menuhandle,0)

```

```

! insert at end of list

```

```

call toolbox(ADDRESMENU,menuhandle,'DRV')

```

```

! add desk accessories

```

```

menuhandle=toolbox(GETMENU,30) ! get 2nd menu
call toolbox(INSERTMENU,menuhandle,0)

```

```

! insert at end of list

```

```

menuhandle=toolbox(GETMENU,31) ! get 3rd menu
call toolbox(INSERTMENU,menuhandle,0)

```

```

! insert at end of list

```

```

menuhandle=toolbox(GETMENU,32) ! get 4rd menu
call toolbox(INSERTMENU,menuhandle,0)

```

```

! insert at end of list

```

```

call toolbox(DRAWMENUBAR) ! display menu bar

```

```

* Create the window. the SETPORT call allows FORTRAN I/O
* to take place: The window is created using a resource and
* the GETNEWWINDOW toolbox call.

```

```

d_window = toolbox(PTR,d_record)
d_window = toolbox(GETNEWWINDOW,1,d_window,-1)
call toolbox(SETPORT,d_window)

```

```

* constraints on window dragging

```

```

rect(1) = 30
rect(2) = 1
rect(3) = 350
rect(4) = 510

```

```

* build a text edit record

```

```

trect(1)=50
trect(2)=4
trect(3)=245
trect(4)=405
hte=toolbox(TENEW,trect,trect)

```

```

* main event processing loop

```

```

do

```

```

* handle system jobs

```

```

call toolbox(SYSTEMTASK)

```

```

* handle events

```

```

if (toolbox(GETNEXTEVENT,eventmask,eventrecord)) then

```

```

select case (what)
case (1)

! mouse down
mouseloc = toolbox(FINDWINDOW,where>window)

! locate mouse
if (mouseloc=1) then
 mouse(0)=where(0)
 mouse(1)=where(1)
 call menus(mouse,d_window,hte)
! in menu bar
else if (mouseloc=2) then
 call toolbox(SYSTEMCLICK,eventrecord>window)
! in system window
else if (mouseloc=3) then
 call toolbox(SELECTWINDOW>window)
! in content region
else if (mouseloc=4) then
 call toolbox(DRAGWINDOW>window,where>rect)
! in drag region
else if (mouseloc=6) then
 if (toolbox(TRACKGOAWAY>window,where)) then
! in goaway region
 call toolbox(HIDEWINDOW>window)
 menuhandle=toolbox(GETMHANDLE,32)
 call toolbox(ENABLEITEM,menuhandle,1)
 call toolbox(DISABLEITEM,menuhandle,2)
 end if
end if
case (6)

! update event
call Up_window(d_window)
case (8)

! activate event
if (mod(modifiers,2) .ne. 0) then
 call toolbox(TEACTIVATE,hte)
 menuhandle=toolbox(GETMHANDLE,31)
 call toolbox(DISABLEITEM,menuhandle,1)
 call toolbox(DISABLEITEM,menuhandle,3)
 call toolbox(DISABLEITEM,menuhandle,4)
 call toolbox(DISABLEITEM,menuhandle,5)
 call toolbox(DISABLEITEM,menuhandle,6)
 call toolbox(SETPORT,d_window)
else
 call toolbox(TEDEACTIVATE,hte)
 menuhandle=toolbox(GETMHANDLE,31)
 call toolbox(ENABLEITEM,menuhandle,1)
 call toolbox(ENABLEITEM,menuhandle,3)
 call toolbox(ENABLEITEM,menuhandle,4)
 call toolbox(ENABLEITEM,menuhandle,5)
 call toolbox(ENABLEITEM,menuhandle,6)
end if
case default

! ignore other events
end select
end if
repeat

```

```

! repeat for another event
end

*
* menus: a mouse down event was detected in the menu area
*

subroutine menus(mouse,d_window,hte)

implicit none
include desk.inc
include dialog.inc
include event.inc
include menu.inc
include quickdraw.inc
include misc.inc
include textedit.inc
include window.inc

integer*4 toolbox
integer*2 where(2) ! mouse location from event record
integer d_window ! d window pointer
integer window ! general window pointer
integer*4 dlg ! dialog pointer
integer*2 ditemh ! item hit in dialog
integer*4 menuhandle ! menuhandle pointer
character*256 name
integer*4 refnum,item4,hte
integer*2 mouse(2) ! where() field of eventrecord()

*
* variables for making menu selections
*

integer*2 menuselection(2) ! menu selection information
integer*4 menudata ! for use left of equals sign
equivalence (menuselection,menudata)

*
* Menu selection constants:
*

integer Apple ! menu
integer About ! "Apple" menu selections
integer File ! menu
integer Quit ! "File" menu selections
integer Edit ! menu
integer Undo,Cut,Copy,Paste,Clear ! "Edit" menu
integer Options ! menu
integer Smsg,Hmsg ! "Options" menu selections

parameter (Apple=29,File=30>Edit=31,Options=32)
parameter (About=1)
parameter (Quit=1)
parameter (Undo=1,Cut=3,Copy=4,Paste=5,Clear=6)
parameter (Smsg=1,Hmsg=2)

*
* Start of Subroutine
*

menudata=toolbox(MENUSELECT,mouse)
! get selected menu data

item4=menuselection(2) ! convert to 4 bytes
select case (menuselection(1)) ! which menu?

case (Options) ! "Options" menu
 menuhandle=toolbox(GETMHANDLE,Options)
 ! get "Options" handle

```

```

select case (menuselection(2))
 ! "Options" menuitem selected

 case (Smsg)
 call toolbox(SHOWWINDOW,d_window)
 call toolbox(SELECTWINDOW,d_window)
 call toolbox(DISABLEITEM,menuhandle,Smsg)
 call toolbox(ENABLEITEM,menuhandle,Hmsg)
 case (Hmsg)
 call toolbox(HIDEWINDOW,d_window)
 call toolbox(DISABLEITEM,menuhandle,Hmsg)
 call toolbox(ENABLEITEM,menuhandle,Smsg)
 case default
end select
case (File)
 ! "File" menu selected
 menuhandle=toolbox(GETMHANDLE,File)
 ! get "File" handle

select case(menuselection(2))
 ! "File" menuitem selected
 case(Quit)
 stop
 case default
end select
case(Apple)
 ! "Apple" menu selected
 menuhandle=toolbox(GETMHANDLE,Apple)
 ! get "Apple" handle
select case(menuselection(2))
 ! "Apple" menuitem selected
 case(About)
 ! About item selected
 dlg=toolbox(GETNEWDIALOG,200,0,-1)
 do
 call toolbox(MODALDIALOG,0,ditemh)
 if (ditemh=3) goto 100
 repeat
100 call toolbox(DISPOSEDIALOG,dlg)
 case default
 ! Desk Accessory selected
 call toolbox(GETITEM,menuhandle,item4,name)
 refnum=toolbox(OPENDSKACC,name)
 end select
case (Edit)
 if (.not. toolbox(SYSTEMEDIT,item4-1)) then
 ! edit not in dsk acc
 call toolbox(SETPORT,d_window)
 select case (menuselection(2))
 case (Cut)
 call toolbox(TECUT,hte)
 case (Copy)
 call toolbox(TECOPY,hte)
 case (Paste)
 call toolbox(TEPASTE,hte)
 case default
 ! ignore other edit menu items.
 end select
 end if
case default
 ! just playing with the mouse
end select
call toolbox(HILITEMENU,0)
! unhilites menu
end

```

```

! end of menus subroutine
*
* MSG: clear the window and position the pen at 3,12 (v,h)
* and put up message
*
subroutine MSG

implicit none
integer*4 toolbox
include desk.inc
include dialog.inc
include event.inc
include menu.inc
include quickdraw.inc
include misc.inc
include textedit.inc
include window.inc

call toolbox(MOVETO,30,30)
call toolbox(TEXTFONT,1)
call toolbox(TEXTSIZE,12)
call toolbox(TEXTFACE,1)
type "Porting to the Mac"
call toolbox(MOVETO,100,100)
type "A series devoted to porting"
call toolbox(MOVETO,100,130)
type "fortran program to the Mac"
call toolbox(MOVETO,100,160)
type "from mainframe and minicomputers"

end
*
* subroutine to update window after event processed using
* window
*
subroutine Up_window(d_window)

implicit none
include desk.inc
include dialog.inc
include event.inc
include menu.inc
include quickdraw.inc
include misc.inc
include textedit.inc
include window.inc
integer*4 toolbox,d_window

call toolbox(BEGINUPDATE,d_window)
call toolbox(SETPORT,d_window)
call MSG
call toolbox(ENDUPDATE,d_window)

end

```

\*  
 \* Edit date: 08/13/85  
 \* This is a RMAKER file initially generated by the EDITOR.  
 \*

Shell  
 APPLMEMM

\* Include resource code generated by MacFortran  
 INCLUDE Shell Prog

\*  
 \* Set up id string  
 TYPE MEMM = GNRL  
 ,0 ;; id name is signature  
 .I  
 11  
 .P  
 MARKMCBRIDE

\*  
 \* Set up file reference  
 TYPE FREF  
 ,4000 ;; resource id  
 APPL 0 ;; file type, id of icon

,4001 ;; resource id  
 WIND 1 ;; file type, id of icon

\*  
 \* Set up bundle resource  
 \*

TYPE BNDL  
 ,4000 ;; resource id  
 MEMM 0 ;; bndl owner  
 ICN# ;; resource type  
 0 4000 1 4001 ;; local ID 0 maps to resource ID 128  
 FREF ;; resource type  
 0 4000 1 4001 ;; local ID 0 maps to resource ID 128

Type ICN# = GNRL ;; icon list for application  
 ,4000 ;; resource ID  
 .H  

|          |          |          |          |
|----------|----------|----------|----------|
| 00000000 | 000025FD | FFFFFF00 |          |
| FFFFFFE0 | 02001505 | FFFFFF00 | 000007FF |
| B9F1D200 | 00000D05 | FFFFFF00 | 000007FF |
| B9F1D200 | 02AAAD05 | FFFFFF00 | 000007FF |
| B9F1DE00 | 00000DFD | FFFFFF00 | 000007FF |
| FFFFFFE0 | 00001401 | FFFFFF00 | 000003FE |
| 80001000 | 00002405 | 02000000 |          |
| 80001000 | 0000047D | 02000000 |          |
| BFFFD000 | 00000401 | 02000000 |          |
| A4045000 | 00000401 | 02000000 |          |
| A4045000 | 000007FF | 02000000 |          |
| A4045000 | 000003FE | 020007FF |          |
| FFFFFF00 | *        | 020007FF |          |
| 02000000 | 00000000 | 020027FF |          |
| 00000000 | FFFFFFE0 | 020037FF |          |
| 02000000 | FFFFFFE0 | 02003FFF |          |
| 00000000 | FFFFFFE0 | 03FFFFFF |          |
| 02000000 | FFFFFFE0 | 00003FFF |          |
| 000007FF | FFFFFFE0 | 000037FF |          |
| 02000401 | FFFFFF00 | 000027FF |          |

\* This is the definition for the MENU.

Type MENU  
 ,29  
 \14  
 About Shell  
 (-

,30  
 File  
 Quit

,31  
 Edit  
 (Undo  
 (-  
 (Cut  
 (Copy  
 (Paste  
 (Clear

,32  
 Options  
 (Show  
 Hide

\* Definition to setup window  
 \*

Type WIND  
 ,1  
 Shell Window  
 50 20 250 400  
 Visible GoAway  
 4  
 0

\* Definition to setup dialog box  
 Type DLOG  
 ,200  
 ;; no name for this resource  
 50 50 212 404  
 Visible NoGoAway  
 1  
 0  
 200

Type DITL  
 ,200  
 3  
 StatText Disabled  
 14 38 34 316  
 An application shell in MacFortran

StatText Disabled  
 94 20 114 339  
 Mark E. McBride August 15, 1985

BtnItem Enabled  
 126 136 141 182  
 ok





# *Lisp Listener*

## *An Object Oriented Language*



Andy Cohen  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 6

Welcome to The Lisp Listener Window! This is the first of what we hope will become a regular column on programming Lisp on the Apple Macintosh. Lisp stands for List Programming and it is used for a very special type of computer programming in a field typically referred to as Artificial Intelligence, or AI. Over the past five years, AI has become a very popular and controversial topic. It has roots in both Computer Science and Cognitive Psychology. The psychologists found ways of describing human mental processes, and the computer scientists have programmed these processes into computers, thereby getting the computer to perform human-like processing tasks.

Computers however, are definitely not one-to-one analogies to people. Computers process data, while human processes deal with concepts. Lisp was designed specifically to handle the coding of these concepts by providing an object oriented, programming environment. This development has provided us with programs that generate information by mimicking the same form of inductive reasoning used by people. In the case of specialized knowledge domains, these programs have been called Expert Systems. I will eventually try to emphasize in Lisp examples which take advantage of Lisp's differences from the other languages covered in this publication, primarily its ability to manipulate objects, rather than just data.

### **Is Lisp on the Mac Serious?**

A friend asked me a little while ago, will we be able to program serious AI applications using Lisp? My first reaction was that we can't. The smallest development system for an AI application has quite a bit more random access memory (from two to five Mbytes) and usually uses a hard disk with more than 30 megabytes of space. The best we can hope for in the Macintosh environment (at least at the time this article is written!) is one megabyte of RAM and ten megabytes on a hard disk using the Mac XL. However, I also noted that sometimes the application is only developed using Lisp and then when finalized, the code is translated into a lower level language, such as Fortran. This recoding is done for speed and compactness.

My secondary feelings were yes, we probably can develop an AI application on the Mac to some extent, but the program will have to be translated into Pascal, C or even 68K Assembly. As it turns out, the version of Lisp coming from Expertelligence will allow us to do quite a bit with just 512K and two floppies. I'll describe this package in detail later. Will we be able to use the Mac as an AI development machine? I

am still skeptical, but I have a feeling that in time it just might be possible. If you are asking, why would somebody program Lisp on the Mac at this time? I can only say one should use Lisp on the Mac for educational purposes. Heck, educational purposes are more than enough, considering that a complete Lisp development system can cost over \$75,000. Also consider that just a couple of years ago, having Lisp on a microcomputer was almost as fictional as verbally asking Mac to open the pod bay doors!

### **Different Lisps**

Lisp comes in many different versions just like any other computer language. There is Zetalisp, Interlisp, MacLisp (not Macintosh) and more I'm sure I've never heard of. The versions of Lisp that are or will be available on the Macintosh are very close to Common Lisp. An excellent text for Common Lisp is "Lisp" 2nd ed. by Winston and Horn. All that is discussed in this article regarding Common Lisp follows Winston and Horn.

At the time of this writing, only one version of Lisp is available. It is called XLisp, and was developed by David Betz. It was programmed in C and was considered by Betz as an experimental language. There are two versions of XLisp that I have seen on the Mac; versions 1.2 and 1.4. Both versions of XLisp are interpreted. Only version 1.4 however, can handle object oriented programming. Unfortunately, XLisp 1.4 takes up most of the available memory on a 128K Mac. Either version, however, will work fine on a 128K Mac for demonstrating some of the arithmetic functions in Common Lisp syntax. The biggest advantage of XLisp is that it is Public Domain software, in other words, free. Look for it on CompuServe's MAUG or to your local users' group. Chances are it is available in one or more versions.

### **Lisp and Arithmetic**

Enough small talk (oops, sorry about that!): now let's take a first look at Lisp. The most fundamental object in Lisp is an *Atom*. The atom is used in groups to produce *Lists*. Atoms and lists make up what are called *expressions*.

Okay, let's get a little more detail. A *procedure* specifies how something is performed. For example "+" or plus sign is a procedure. The "+" by itself is called a *primitive*. Numbers or symbols are referred to as *arguments*. Therefore the numbers 2.56 and 8.34 are arguments. These elements or atoms can be placed into a List:

(+ 2.56 8.54)

The procedure "+" will then operate on the arguments 2.56 and 8.54 to produce 11.1. A bunch of procedures which work together are a program. More examples of lists are the following:

```
(+ 2 2)
(* 103 346)
(/ 35 46)
```

If you have XLisp, try typing these in yourself. The answers are produced when the carriage return is made, then displayed beneath the list. I used bold lettering to indicate the output.

```
(+ 2 2)
4
```

See. Isn't that easy? Let's make things more interesting. Procedures can also be more complex, e.g. MIN and MAX (If you are trying this with XLisp be sure to use the lower case):

```
(MIN 53 37 95 23)
23
(MAX 74 37 49 20)
74
```

MIN will print the smallest argument in the list. MAX will print the largest.

One can also use primitives typically used with multiple arguments with singular arguments.

```
(- 7)
-7
(- -3)
3
```

In the first case, the negative or subtract symbol turns the argument into a negative equivalent. In the second the "-" turns the negative argument into a positive just as  $-1*-3=3$ .

Lists may contain other lists.

```
(- (* 4 9) (+ 8 32))
-4
```

The equivalent of the above is as follows:

```
(- 36 40)
-4
```

As we can see, the list is organized and performed in a typical hierarchical manner. A list, therefore, is made up of a left parenthesis, followed by any number of atoms or lists and concluded with a right parenthesis.

### Dissecting Lists

Enough of numbers. Let's start to use symbols as arguments.

```
(A B C)
```

or

```
(Alpha Beta Cookies)
```

Now let's use more sophisticated procedures. CAR and CDR are used to pull arguments out of lists. I'm told CAR stands for "Contents of Address Register" and CDR stands for "Contents of Data Register". I'm sure there is some meaning in these titles with regard to the result, but that's not important right now. What is important is what these procedures do.

```
(CAR '(Alpha Beta Cookies))
Alpha
```

CAR returns the first argument in the list. Note that we are using a list in a list.

```
(CDR '(Alpha Beta Cookies))
(Beta Cookies)
```

CDR returns the list minus the first argument. What if I am looking for the second argument in a list of three? In this case the CAR and CDR procedures can be nested.

```
(CAR (CDR '(Alpha Beta Cookies)))
Beta
```

In the above example CDR produces the list,  
(Beta Cookies)

then CAR produces Beta. I hope you've noticed the single quote after the CDR above as well as in the other samples. This is how Lisp knows that the CDR in the above sample is a procedure and not just another argument. Without the single quote the following would be the result:

```
(CAR (CDR (Alpha Beta Cookies)))
CDR
```

CDR in this case was not considered by Lisp as a procedure. The CAR procedure looked to the first argument, in this case the CDR, and displayed it. After playing a bit with these two procedures and a bunch of arguments one can see ways of making sets of procedures which will perform specific kinds of searches.

I've tried to give a very brief introduction to Common Lisp simply for the purpose of giving some impression of what Lisp looks like. Most of the above can be done using XLisp. Since XLisp is an experimental program and Experlisp from Expertelligence will be a relatively serious Lisp development package, this column will probably deal exclusively with Experlisp. Next month The Lisp Listener Window will give a detailed description of Experlisp as well as discuss composing nested CARs and CDRs, symbolic assignment and a whole bunch of related procedures. In future installments I would like to review detailed examples as well as compare the performance of our little Mac to those of the Lisp machines from Xerox and Symbolics.



# Lisp Listener

## Introduction to XperLisp



Andy Cohen  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 7

Welcome back to MacTutor's Lisp column! By the time this issue is printed, Experlisp from ExperTelligence should be released. For the many of you out there who have yet to see this new version of Lisp, the Lisp Listener Window shall present its first description of Experlisp along with more descriptions of Common Lisp procedures.

### Experlisp

Experlisp consists of a "Listener Window", an editor and a compiler. The Listener Window can act as an interpreter, however it only operates on one line of code. One can use it to try out a simple list and see what the list will do. For multiple line lists or programs, one must write the code using the editor. The editor has its own window or windows, which are called Edit Buffers when untitled. It can have more than one Edit Buffer open at the same time, as in MacDraw and MSWord. The editor consists of all the standard editing operations that appear in most Macintosh text editing programs. Cutting and pasting between files is extremely easy and fast because multiple lists can be displayed simultaneously. After you type in the lists, the "Compile" menu offers a choice of compiling and executing a selected group of code, or the entire program. Once either is selected, the compiler kicks in and compiles the Lisp code to 68000 code. It happens quickly and when completed, the program is initiated. All interaction with the program then takes place in the Listener Window, unless programmed otherwise. A screen dump of the Experlisp display is shown below in Figure 1. (This was taken using the prereleased version. The released version, I'm told does not include the Help or Output menus on the menu bar. Also the "Run" menu was changed to "Compile").

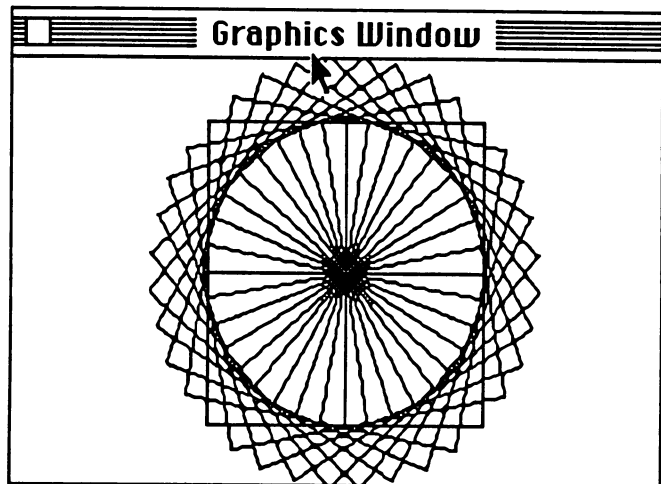


Fig. 1 2-D graphics

Experlisp is a combination of three worlds. First, as I mentioned last month, Experlisp is based on the Common Lisp standard described by Winston and Horn's "Lisp" 2<sup>nd</sup> edition. It also contains elements from Zetalisp, the version of Lisp used on Symbolics workstations. This is no surprise, since Experlisp was developed on a Symbolics machine. Finally (I saved the best part), Experlisp can access the Macintosh toolbox. In fact, one must use the same syntax as shown in Inside Macintosh. Experlisp has the capability to open and close windows, define and create menus on the menu bar and generate all the Macintosh graphics primitives. Experlisp also contains all the graphics features found in Experlogo (also developed by ExperTelligence). One can generate turtle graphics as in any logo, however Experlogo and Experlisp use the term Bunny graphics. This is characterized by instructions which tell a "Turtle" (from the original Logo, which moved a mechanical device which looked like a turtle) or "Bunny" to move in specific directions. Bunny graphics can be two-dimensional, three-dimensional or spherical. Spherical graphics is unusual. In spherical graphics, the lines appear to be drawn upon a three-dimensional globe. Figure 2 shows a box drawn repeatedly while rotated in a circle in 2D; figure 3 is the same box pattern in 3D, and figure 4 is in spherical.

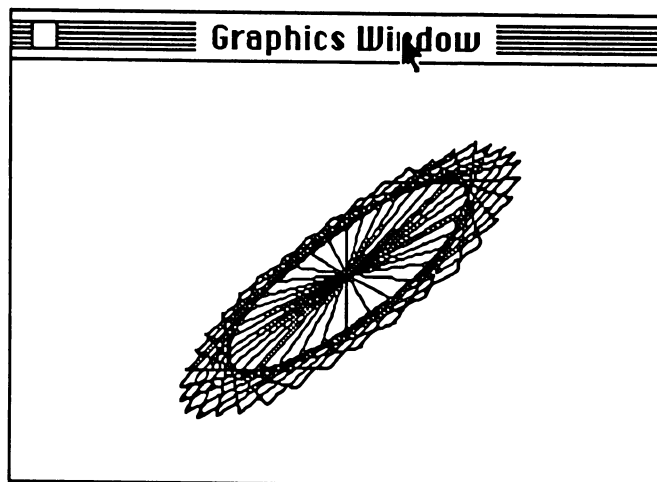


Fig. 2 3-D graphics

Next month I will discuss how these figures were generated.

One of the most exciting features about Experlisp is its speed. It is *fast*. The program operates at a speed comparable to programs written in assembly language and Forth. However, Lisp is significantly easier to learn and it does handle symbol manipulation as we shall eventually see. As I mentioned last month, one might question its useability in AI applications due to the memory size of the Mac.

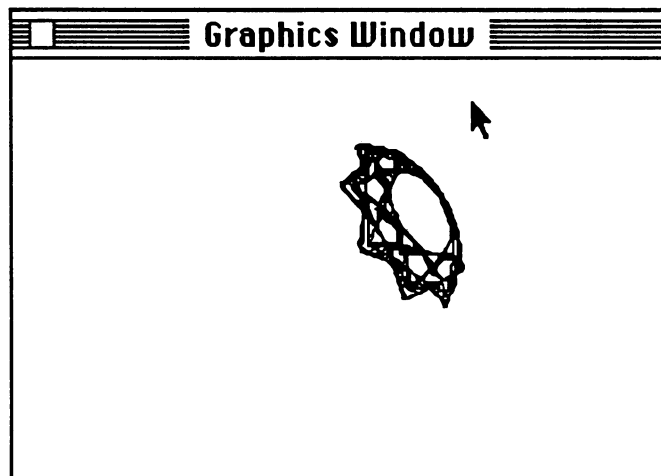


**Fig. 4 Spherical graphics**

ExperTelligence says that the Lisp program code is treated in a virtual manner typical to the Macintosh. Code is loaded when needed. Therefore, the size of your program depends upon disk space rather than RAM. The memory limitations do not affect the program size: instead they affect the size of a function. In Lisp, you can define a function in a way similar to the way Forth generates its syntax. For example:

```
(DEFUN AVERAGE (W X Y Z)
 (/ (+ W X Y Z) 4))
```

The above defines the word **AVERAGE** as the average of a four argument list. Whenever **AVERAGE** is used followed by four numbers it will generate the mean of the numbers.

```
(AVERAGE 7 3 9 5)
;6
```

Experlisp can allow functions up to 250Kbytes in size! Note that my example didn't generate one full Kbyte of code.

It sounds great, however there are imperfections. Even though the compiler generates 68000 code, there is presently no way to execute that code other than via the Experlisp editor and compiler. Stand-alone applications in Experlisp might become a reality soon, however, according to what I have been told by ExperTelligence. The manual states that Experlisp 2.0 will be a "much expanded... developer's version". There is also the question of debugging facilities. In MacPascal and MSBasic 2.0, there are extremely useful tracing commands which show the programmer exactly what the code is doing every step of the way during execution. Unfortunately, Experlisp does not have that. However, that is no surprise since both Macpascal and MSBasic 2.0 are both completely interpreted languages. Remember, Experlisp is compiled. Error messages are displayed in the Listener Window during compilation. Additionally, Experlisp does provide a **TRACE** Macro which shows the values of a function both before and after execution, as well as the ability to break during execution and examine values (although it would be useful to see more than just the effects on arguments at the function level). We hope to provide the code for more indepth tracing in Experlisp sometime in a future installment of Lisp Listener Window.

There is another disadvantage for which there is no easy answer: Experlisp works only on a Macintosh 512 or the XL.

The 128K Mac is too skinny. If this gets you down, try to be patient. The upgrade will more than likely continue to go down in price. One last unfortunate note is cost. Experlisp will list at \$500.00. This high price, I'm told, was due to the development costs of Experlisp and its copy protection scheme. It will (eventually) be protected using a small piece of hardware that connects between the keyboard and the Mac. If Experlisp becomes as popular as I think it will, this number will probably also come down. At this time, Experlisp is a good investment for someone who is very serious about learning and using Lisp for both AI and non-AI programming tasks regardless of the above disadvantages. If I thought otherwise I wouldn't be writing this column!

Last month I mentioned skepticism about whether we might be able to produce a serious AI application on the Mac. This month, after learning more about Experlisp, I'm willing to stick my neck out a bit and say that I feel that a relatively serious application is possible. However, the application will probably be a limited expert system using shallow production rules as opposed to indepth semantic networks. One of the advantages to Experlisp is the ease in which you can build the application's user interface. That in itself places the Mac in the running for real applications using Lisp. Serious AI development will still require at least a hard disk drive along with the external drive. (Time will tell if I'll eat these words).

### More Lisp Procedures

We finished last week with brief descriptions of the **CAR** and **CDR** procedures. The procedures are used in taking lists apart. You can nest these procedures so that longer lists can be dissected. For example:

```
(CDR '(BEANS PEAS CORN CARROTS))
; (PEAS CORN CARROTS)
(CDR '(PEAS CORN CARROTS))
; (CORN CARROTS)
(CAR '(CORN CARROTS))
; (CORN)
```

The above can be written as follows:

```
(CAR (CDR (CDR '(BEANS PEAS CORN- CARROTS))
; (CORN)
```

(All lists following the ";" and in bold are output from the computer) These **CARs** and **CDRs** can be combined into one procedure. The above can also be written as:

```
(CADDR '(BEANS PEAS CORN CARROTS))
; (CORN)
```

Each "A" within the "C" and the "R" represents a **CAR**, while each "D" represents a **CDR**. Any combination is possible. In Experlisp a depth of four combinations are allowed. That is, **CxxxxR**, **CxxxR** or **CxxR**, while **x="A" or "D"**.

Other procedures are used in adding new arguments to a list or for making new lists. These are called **Construct** lists. The **CONS** function (short for constructor) puts a new argument at the beginning of a list.

For example:

```
(CAR (CONS 'x '(y z)))
; x
```

The CONS adds the "x" to (y z ) making (x y z) from which the CAR extracted the "x".

The APPEND function puts the arguments from given lists into one new list. For example:

```
(APPEND '(A B C) '(D E F))
;(A B C D E F)
```

The LIST function takes the arguments from given lists or entire lists and puts them into a new list. For example;

```
(LIST (A B C) (D E F) (G H I))
;((A B C) (D E F) (G H I))
```

The actual value of the LIST function is not apparent since I have not yet discussed assigning values to symbols. One way to perform this assignment is with the SETQ function. SETQ places the value or values of the second argument following the SETQ function and assigns it or them to the first argument. The following demonstrates this:

```
(SETQ VEGETABLES (PEAS BEANS CARROTS))
;VEGETABLES
;(PEAS BEANS CARROTS)
```

After evaluating the first list the symbol, VEGETABLES represents (PEAS BEANS CARROTS). When one types the assigned symbol into the Listener Window (after compiling and running the above list) the assigned values are displayed. Of course this example is extremely simple. In coming installments, the SETQ function will be used with more interesting examples which will better illustrate the object oriented qualities of Lisp. Getting back to the LIST function, the following example shows its real strength in relation to APPEND and CONS:

```
(SETQ J (A B C D))
(LIST J J)
;((A B C D) (A B C D))
or
(LIST 'J J)
;(J (A B C D))
or
(LIST 'J 'J)
```

```
;(J J)
```

```
(APPEND J J)
;(A B C D A B C D)
```

```
(CONS J J)
;((A B C D) A B C D)
or
(CONS 'J J)
;(J A B C D)
```

### Prerelease Caveat

If you have obtained a prerelease version of Experlisp (v0.1), note that the MIN and MAX procedures are not functional. The nested CARs and CDRs won't work either. You might have also noticed that sometime at the end of March, you could not get Experlisp v0.1 to boot up. ExperTelligence has something in the code that checks for the date. Beyond the release date (March 31st), the prerelease should not be functional. (Unless of course you work around the current date..set your calendar back!). There are other problems associated with the prerelease version. The prerelease manual is actually only an incomplete reference manual and does not contain specific details of Experlisp. Also there is some mixture between Zetalisp and the Common Lisp standard. These combinations are not apparent in the prereleased manual, but are in the release. Obviously one should not gamble on using a prerelease Experlisp for any kind of programming effort outside of hacking for fun. If you are without a version of Experlisp, but you do have Xlisp, some of the above described procedures will work. Type them in as shown, but remember to use lowercase.

Next month, the Lisp Listener Window will include sample source listings for generating bunny graphics, windows, menus and various Macintosh toolbox commands.

File Edit Run Help System

Run Selection %R  
Run All

### menuexample

```
(defun box () (dotimes (j 4) (fd 100) (rt 90)))
```

```
(defun spinner () (cs) (
```

```
(defun rband ()
 (print "To exit this
 (to the left of the space
 the mouse button")
 (cs)
 (setq oldx 0)
```

```
Siner
(@gwinclass 'deftop 80)
;sets the default top coordinate of a
graphics window
(@gwinclass 'deftitle "¿Hang Ten?™")
;sets the default title of a graphics
window
(defun sine (freq amplitude)
```

# Lisp Listener

## Functions in Lisp



Andy Cohen  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 8

### Defining Procedures

As you may recall from the first installment of the Lisp Listener, a procedure is a description of an action or computation. A primitive is a predefined or "builtin" procedure (e.g. "+"). As in Forth, Lisp can have procedures which are defined by the programmer. DEFUN, from DEfine FUNction, is used for this purpose. The syntax for DEFUN in Experlisp is:

```
(DEFUN FunctionName (symbols)
 (All sorts of computations which may or may Not use the
 values represented by the symbols))
```

The function name is exactly that. Whenever the name is used the defined procedure associated with that function name is performed. The symbols are values which may or may not be required by the procedures within the defined function. If required, the values must follow the function name. When given, these values are assigned to the symbol. This is similar to the way values are assigned to a symbol when using SETQ. It is easier to see how DEFUN works from an example:

```
;(DEFUN Reciprocal (n)
 (/ 1 n))
;Reciprocal
;(Reciprocal 5)
;2
;(Reciprocal 2384)
;.000419463
```

The word "Reciprocal" is the function name and the numbers following are the values for which the reciprocal (1/n) are found. After the list containing DEFUN is entered and the carriage return is pressed, the function and its title are assigned a location in memory. The function name is then printed in the Listener window.

```
;(DEFUN Square (x)
 (* x x))
;Square
;(Square 5)
;25

;(DEFUN Cubed (y)
 (* y (* y y)))
;Cubed
;(Cubed 5)
;125

;(DEFUN AVERAGE (W X Y Z))
```

```
((+ W X Y Z) 4))
;Average
;(Average 2 3 4 5)
;3.5
```

You might recognize "Average" from last month's Lisp Listener. One might imagine using defined functions inside other defined functions. If it was possible to have variables which have the same values in each procedure, then the version of Lisp used has what is called *dynamic scoping*. In this context the values of the variable are determined by the Lisp environment which is resident when the procedure is called. Experlisp, however, is *lexically scoped*. That means that variable values are local to each procedure. Two defined procedures can use the same labels for variables, but the values will not be considered as the same. Each variable is defined locally. This is in accordance to the Common Lisp standard. Lexical scoping makes it easier to debug someone else's programs. If you don't know what I mean yet, don't worry. This subject will come up again in more detail later.

### Predicates

If no values are required by the defined function then "nil" or an empty list must follow the function name.

```
;(DEFUN Line ()
 (Forward 50))
```

The empty list obviously contains no atoms (I'll describe the above function, "Line" later in the section on bunnies). It is synonymous to the special term nil, which is considered by Lisp as the opposite of T or True. Nil is used in many other contexts.

```
;(cddr '(one two))
;nil
```

In the above, the first cdr returns "two". The second cdr returns nothing, hence "nil". The values of true and false are returned by procedures called predicates. While nil represents a false condition, anything other than nil, including "T", is generally considered true. Please note that I used lowercase letters in the above. ExperLisp recognizes both upper and lowercase. I've been using uppercase only to make it clear within the text when I'm referring to Lisp.

EQUAL is a predicate which checks the equality of two arguments. Note the arguments can be integers or symbols. If the two arguments are equal then "T" is returned. If they are not equal then "nil" is returned.

```
;(EQUAL try try)
;T
```

```
;(EQUAL 6732837 6732837)
;T
```

```
;(EQUAL 6732837 6732833)
;nil
```

```
;(EQUAL First Second)
;nil
```

ATOM checks to see if its argument is a list or an atom. Remember, the single quote is used to indicate that what follows is a not evaluated as in the case of a list. Symbols are evaluated.

```
;(ATOM 'thing)
;T
```

```
;(ATOM thing)
;nil
```

```
;(ATOM (A B C D))
;nil
```

In the first of the above 'thing is an atom due to the single quote. In the second, thing is considered a symbol. A symbol is evaluated and contains a value or values as a list. In the third, (A B C D) is obviously a list.

LISTP checks if its argument is a list.

```
;(LISTP '(23 45 65 12 1))
;T
```

```
;(SETQ babble '(wd ihc wi kw))
;(LISTP babble)
;T
;(LISTP 'babble)
;nil
```

```
;(LISTP 'thing)
;nil
```

One interesting observation is that nil is both an atom and a list, ()=nil. Therefore ATOM and LISTP both return true for nil.

```
;(ATOM ())
;T
```

```
;(LISTP ())
;T
```

When one needs to know if a list is empty, NULL does the job.

```
;(NULL good)
```

```
;nil
```

```
;(NULL (X Y Z))
;nil
```

```
;(NULL ())
;T
```

```
;(NULL nil)
;T
```

NUMBERP checks if the argument that follows is or represents a number rather than a string.

```
;(NUMBERP 56.887)
;T
```

```
;(NUMBERP fifty-six)
;nil
```

```
;(SETQ fifty-six '(56))
;(NUMBERP fifty-six)
;T
```

Now for a real slick one. MEMBER tests whether or not an argument is a part of a list. An easy demonstration follows:

```
;(MEMBER 'bananas (apples pears bananas))
;(apples pears bananas)
;(MEMBER 'grapes (apples pears bananas))
;nil
```

When the argument is a member, then the contents of the list are given. If not then nil is returned. MEMBER also checks symbols of lists.

```
;(SETQ fruit '(apples grapes pears))
;(MEMBER 'grapes fruit)
;(apples grapes pears)
;(MEMBER 'banana fruit)
;nil
```

EVENP tests to see if an integer is even and MINUSP checks if an integer is negative. ODDP and PLUSP are not needed since they are simply opposite of the first two.

```
;(EVENP 2)
;T
```

```
;(EVENP (- 806 35))
;nil
```

```
;(MINUSP 25)
;nil
```

```
;(MINUSP (-34 86))
;T
```

In the second and fourth examples above the lists contained within are calculated prior to MEMBERP

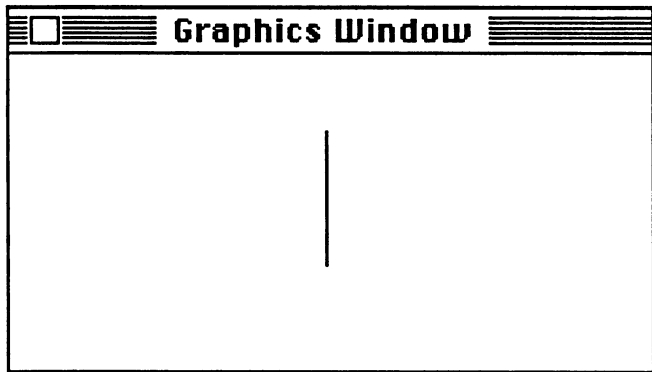
evaluation. (806-35=771 & 34-86=-52. There's a few more simple predicates such as NOT, <, >, and ZEROP. I'll discuss them along with conditionals next month. Now for something completely different.

## Bunny Graphics

If you've ever learned Logo, the concept of Bunny graphics should sound familiar. As mentioned last month, the Bunny is Expertelligence's version of the Turtle. All one needs to do in order to make a Bunny move is to tell it to. FORWARD X initially moves the Bunny upwards on the screen for 'X' display pixels. A negative number initially moves it down. When one enters the following in the Listener window,

```
;(FORWARD 50)
```

the default graphics window (I'll discuss windows in more detail very soon in future installments) is then opened and the following is drawn:



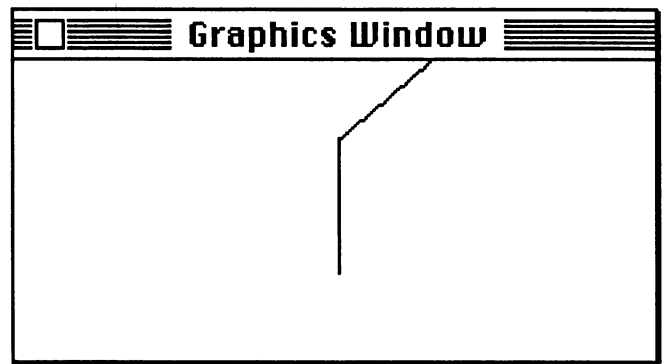
RIGHT X aims the front of the line to the right by X degrees. If one then uses forward again the line moves in a different direction. For example:

```
;;(RIGHT 50) (FORWARD 50))
```

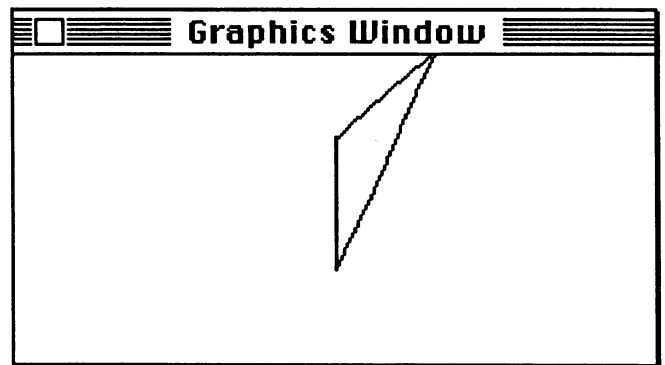
or better yet

```
;(DEFUN Line
 (RIGHT 50) (FORWARD 50))
;Line
;(Line)
```

After a line is moved, the end of the line remains where it was. If one made the Bunny move again the beginning of the new line would begin where the old left off. The original starting point is the graphics window default home position. This position is in the center of each graphics window when the window is first created. In order to return the Bunny to the original starting point one must use HOME.



```
;(HOME)
```



The following produces a much neater triangle:

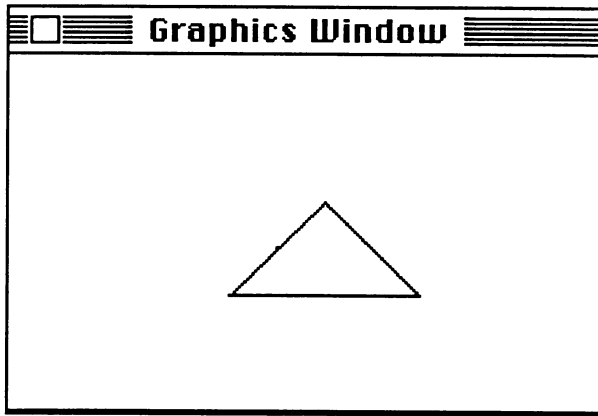
```
(DEFUN Triangle ()
 (Penup) (Left 45)
 (Forward 10) (Pendown)
 (Right 90) (Forward 25)
 (Right 90) (Forward 50)
 (Right 135) (Forward 71)
 (Right 135) (Forward 25))
```

After the above is typed into the edit buffer the "Compile All" selection should be chosen from the Menu Bar. The source code in the Edit Buffer quickly inverts to white letters on a black background as if the whole file was selected for a moment. The function name "Triangle" is then printed in the Listener window. If the user enters the following in the Listener Window a different triangle is drawn in the default Graphics Window:

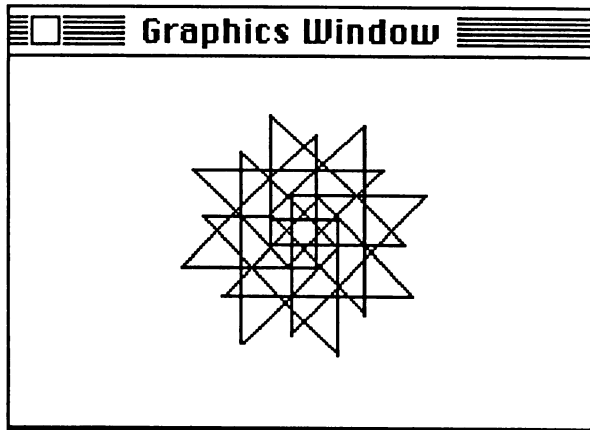
```
;(Triangle)
```

If you Look at the in Triangle you will see a couple more Bunny commands. LEFT does the same as RIGHT but in the opposite direction. PENUP raises the Bunny's pen so that when the Bunny moves no lines are drawn. PENDOWN returns the Bunny to the drawing orientation. The first line of code in "Triangle" puts the Bunny off the Home position so that the drawn triangle will be centered on the screen. As





mentioned earlier, the orientation of the bunny remains. The last line of code in "Triangle left the Bunny aimed at about 1:00 rather than the initial position, 12:00. If we were to make "Triangle" execute ten times without eliminating the Graphics Window the following would result:



In getting "Triangle" to execute recompilation of the code in the edit buffer is not necessary. To get the above one can type the function name into a list ten times within the Listener window. The following however, is easier:

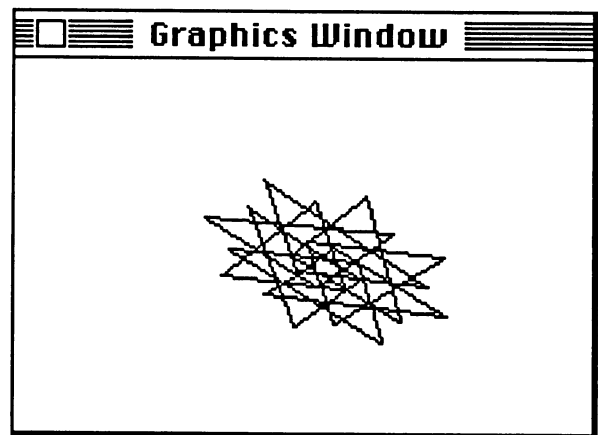
```
;(Dotimes (a 10) (Triangle))
```

DOTIMES is very similar to the FOR...NEXT looping routine in BASIC. I'll discuss it next month in a description of iteration and recursion in ExperLisp.

If we wanted to use a three dimensional bunny then the following would be added before "Triangle" in the Edit Buffer window:

```
(SETQ curbun (new3dbun))
(Pitch 30) (Yaw 45) (Roll 50)
```

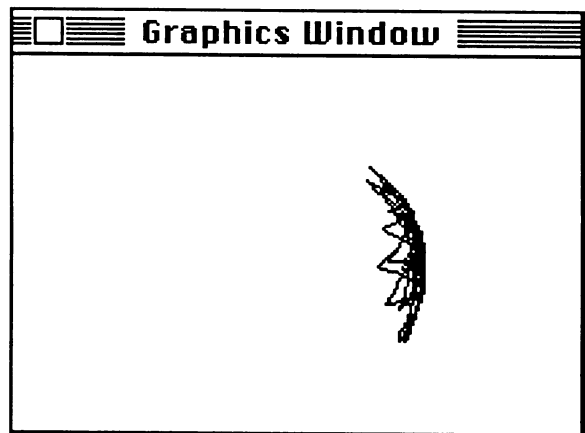
Something like the following is drawn after the source code is recompiled and "(Triangle)" is entered into the Listener Window:



CURBUN is a special symbol in ExperLisp which always refers to the Bunny cursor. NEW3DBUN is a special term which always changes CURBUN. The default Bunny is 2 dimensional. If one wanted the Spherical Bunny then the following would be entered into the beginning of the first version of "Triangle":

```
(SETQ curbun (newsbun))
```

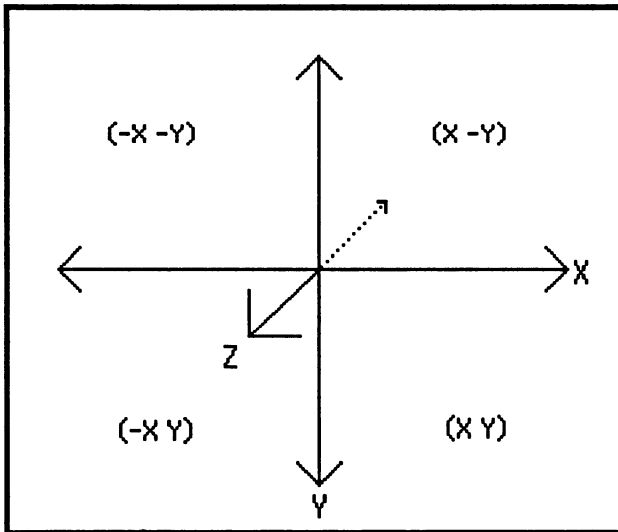
This would then produce what follows:



In order to have the above drawn in a different orientation, different Bunny direction would be required. Windows, two and three dimensional Bunny graphics and toolbox graphics use the same X,Y coordinate system. Home is 0,0. Dual negative coordinates are situated towards the upper left corner. Dual positive coordinates are situated towards the lower right corner. The range is +32767 to -32768 for each dimension. In ExperLisp one can sometimes use the third dimension, as in the 3D sample of "Triangle". Negative Z values are behind Home, while positive Z values are in front. The following illustrates the coordinate system in ExperLisp:

### Compiler Information

The ExperLisp disk contains three essential files; Compiler, LispENV and Experlisp. Compiler is not actually



the entire Lisp compiler. It contains the information needed in generating all of the higher level Lisp syntactics, such as the Bunny graphics. LispENV stands for Lisp Environment and it is simply a duplication of Compile. LispENV contains information on how the Macintosh memory was organized by the programmer and ExperLisp during the previous session. It also contains information on the system configuration such as the number of disk drives, the amount of memory, etc.

Sometimes LispENV can be messed up (i.e. by changing the variable table). When this happens one might not be able to start ExperLisp. In this case LispENV should be removed from the disk. Afterward, when ExperLisp is opened, Compiler generates a new LispENV. Compiler is not needed on the disk unless the LispENV is ruined. Deleting it will provide 100K more space on the disk. Before eliminating it from the disk however, be sure you have a backup as it is an essential file. The Experlisp file contains the assembly language routines which represent the lower level Lisp routines like CAR and CDR. It also allows access to the Macintosh toolbox routines and contains the Listener Window. One opens the Experlisp file in starting a programming session with ExperLisp. Another file on the disk is automatically loaded and activated when Experlisp is booted. It is labeled \*lispinit. The contents of this file can be added to so that when one boots up ExperLisp a program can be automatically executed. It can also do automatic configurations. However the contents of \*lispinit should not be changed since it configures the Macintosh memory for Exper- Lisp.

Next month I'll discuss a few more predicate procedures. I also hope to start discussing iteration, recursion and conditionals. If there is enough room left over I might also begin discussing how to access the toolbox graphics.



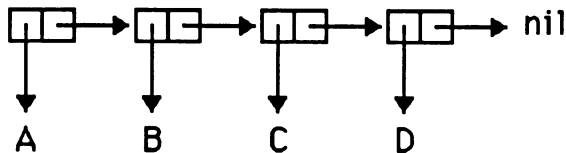
# Lisp Listener

## Cons Cells and Quickdraw!

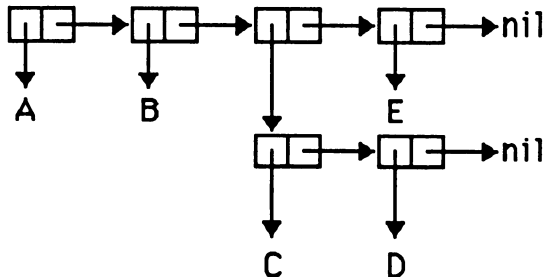


Andy Cohen  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 9

The items within a list have pointers to values and to other items. These list inhabitants are referred to as *cons cells*. A list can be illustrated in terms of its cons cells. For example, (A B C D) can be illustrated as follows:



(A B (C D) E) looks like this:



Each pointer for each cell is a stored address of the location in memory of either the value of the cons cell or the next cons cell. It follows then that the cons cell has, two halves. The first half, which points to the value, contains the contents of the address register. While the other half which points to the next cons cell contains the contents of the decrement register. CAR and CDR, remember?. If you are wondering about the similarity between these acronyms and the functions discussed in this column three months ago, you are right, there is a relationship. However, this relationship is not necessarily significant, actually they are analogies. CAR returns the first part of a list or in the context of the cons cell, the value. CDR returns the rest ( or the next cons cell) minus the first. The procedures could just as easily have been called something more meaningful and they can be if one assigned one of these procedures to a preferred name using DEFUN. I am not bringing this subject up as an aid in understanding these lower level functions. My point is to mention that the number of cons cells a machine can handle is a good estimate of the capabilities of the machine. With a large "AI" workstation such as the Symbolics 3600 (upon which much of Experlisp was developed) one can expect to have enough memory to utilize in various ways approximately 30,000 cons cells. Using a 512K Macintosh with Experlisp one can obviously expect much less. Added into the latest version of Experlisp

(v1.02) there is an undocumented procedure called FREECONS. Contact Expertelligence for an update. Entering FREECONS into the Listener Window returns the number of available cons cells. On a 512K Mac with ExperLisp booted up and with only an untouched 'lispinit file compiled, FREECONS returns 5912. With two megabytes in an XL or a suped up Mac 512 one should expect to get at least 23648 cons cells. Not bad considering the Symbolics 3600 costs in excess of \$100,000.

### More Predicates

In our discussion on predicates from last month there were a few more predefined predicates provided in Experlisp that were not mentioned. Instead of going through each of them the following table contains most of the predicates listed in the Experlisp reference manual:

### ExperLisp Predicates

AND  
ARRAYP  
ATOM  
BOUNDP  
BUILTINP  
CHARACTERP  
CONSP  
COPY-LIST  
EQUAL  
FUNCTIONP  
LAMBDA P  
LISTP  
LOCATIVEP  
MACROP  
MEMBER  
NAND  
NEQ  
NOT  
NULL  
NUMBERP  
OR  
SPECIALP  
SYMBOLP  
TAILP  
TYPE-OF  
VARIABLEP  
=  
>  
<

As you may recall a predicate is simply a procedure which tests the value or characteristics of an argument. It always returns what could be thought of as either true or false. Some of the more interesting predicates in the above list include ARRAYP which tests to see if an array was set up and assigned an address, BUILTINP which tests whether or not a function is a valid primitive and returns its address if it is, and TAILP which tests whether or not a smaller list is at the tail end of a larger list. While these predicates are already defined when Experlisp is opened, there is certainly no reason why one would never want to create their own predicates via DEFUN. For example:

```
;(DEFUN NAMEP (a)
 (EQUAL "Andy" a))

;(NAMEP "Fred")
;nil
;(NAMEP "Andy")
;t
```

When defining a predicate, the "P" at the tail end of the function name is a typical way to identify it as such. However, predicates are not necessarily always written with this syntax. For example, EQUAL tests for equality between groups of alphanumeric characters (the "=" is used for numerals).

### Conditionals

As in most other computer languages there must be a way of performing boolean logic. The terms "If-Then" are typical since they illustrate the logical form. This form is consistent within Experlisp although the syntax is a little different. The COND procedure performs this "If-Then" logic. The typical syntax is as follows:

(COND ((**"IF"** clause) **"THEN"** values, lists or procedures)

In BASIC one will find many different varieties of "If-Then" commands such as "If-Then-Else", "If-or-then-else" or "If-and- then-else", etc. COND provides us with any variety of conditional statement required. Any appropriate predicate, including one defined by the user, may be used within the clause of the COND. ExperLisp looks through each of these tests or clauses within the COND procedure until one returns anything other than nil. If all the clauses return nil, nil is returned. The following is an example:

```
(Defun Which (x)
 (cond ((= 1 x) '(the number is one))
 ((= 2 x) '(the number is two))
 ((= 3 x) '(the number is three)))

;Which
;(Which 2)
;the number is two
;(Which 4)
;nil
```

The above is invoked when the function name and the appropriate form of parameter are evaluated. In the first list inside the COND procedure "(= 1 x)", x is tested for equality with the number one. Since the number two was sent this predicate returns nil. The next list containing "the number is one" is skipped, that is it is not returned and the next line's clause is then evaluated. The parameter, x is then tested for equality with the number two. Since the number two was sent, this list returns "t" and the following list containing "the number is two" is returned in the Listener Window. The third line containing the clause "(= 3 x)" is never evaluated. Once Experlisp gets a nonil from an "IF" clause, the "Then" atoms, lists, or procedures are returned and the COND procedure is completed.

Last month the predicate "MEMBER" was discussed. This predicate, as you may recall, tests to see whether or not an argument is a member of a list. The following sample of COND contains this predicate:

```
(Setq vegies '(carrots, peas, eggplants, broccoli))
(DEFUN foodfun (x)
 (COND ((member x vegies) print "it is a member")
 (not (member x vegies) print "It is not a member")))

;(carrots, peas, eggplants, Broccoli)
;foodfun
;(foodfun apple)
;It is not a member
;(foodfun peas)
;it is a member
```

In the above, the list "(carrots, peas, eggplants, Broccoli)" is assigned as a value to the symbol "vegies". Whenever the term "vegies" is then used (without the single quote, since the single quote will prevent ExperLisp from evaluating the symbol. Symbols have to be evaluated in order to get to their values. Lists or atoms are not evaluated, hence the single quote) it will refer to this list of vegetables. When the defined procedure foodfun is evaluated, along with an appropriate parameter, it passes control to the COND procedure. In this procedure the passed parameter's value is evaluated to see if it is contained within the list represented by the symbol vegies. In the first call to foodfun, the parameter "apple" is not a member of vegie. MEMBER returns nil and control is passed to the next "IF" clause in the COND procedure. This next line is a little more complicated. The parameter is again tested for its membership within vegies, however there is also an evaluation for the return of a nil or a nonnil. NOT returns "t" for nil and "nil" for a nonnil. Since "apple" is still not a member of vegies the nil is returned. NOT then returns a "t" and then the following atoms, lists or procedures are returned. In this case "It is not a member" is printed in the Listener Window. In the second call to foodfun the parameter value is "peas". This time the first list evaluates its membership in vegies and returns "t". The arguments, lists or procedures following this evaluation are then invoked. In this second call to "foodfun", "it is a member" is printed.

Note that in the two above samples two different forms

of output are used. In "Which" the contents of the list are returned. In "foodfun" the PRINT procedure prints the following text inside the Listener Window. One should also note that the second clause in "foodfun" could be performed just as well with the following:

```
(t (print "It is not a member"))
```

The clause is "t" which forces COND to return "it is not a member". In the above one makes it impossible for COND to return "nil".

ExperLisp provides another simpler form of conditional which can be used instead of COND for situations which don't require multiple clauses. IF acts in the same way as COND except it only has one If-Then-Else sequence. The following demonstrates the IF syntax:

```
(DEFUN Iftry (x)
 (IF (= 5 x) '(the number is five) '(the number isn't five)))
```

```
;(Iftry 2)
;(the number isn't five)
;(Iftry 5)
;(the number is five)
```

IF contains three parts; the "If" condition, the "Then" or true return and the "Else" or false return. Note, that the apostrophe (option "]") was used in the word "isn't". If the single quote was used as an apostrophe by accident the following would result:

```
;(Iftry 2)
;(the number isn (quote t) five)
```

ExperLisp reads the single quote as the special symbol used as described above.

### Quickdraw!

A language for the Mac without access to the Quickdraw routines in ROM cannot be taken seriously. Obviously, ExperLisp can utilize these routines. They are used with the same syntax as that described in Inside Macintosh. For example the following draws a rectangular figure:

```
(Paintrect '(top left bottom right))
```

To draw a circular object or oval try the following:

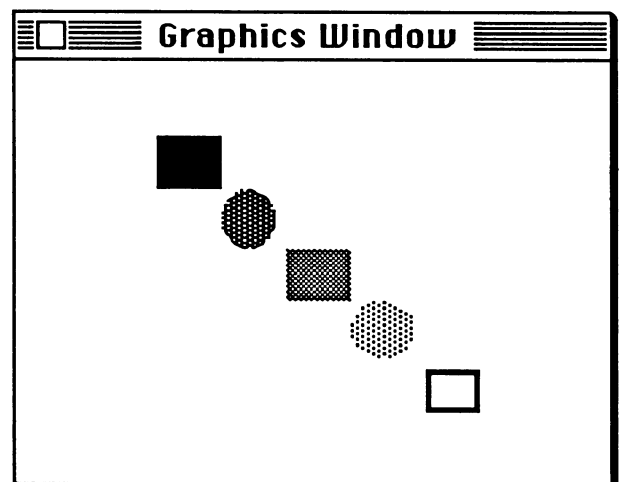
```
(Paintoval '(top left bottom right))
```

Top left and bottom right refers to the coordinates within the specified or default graphics window. These coordinates are as described in last month's column. *Top* corresponds to the vertical coordinate, *left* corresponds to the horizontal coordinate. The coordinate order for the bottom points are consistent.

PENPAT is an ExperLisp function which sets the graphic pen output to one of five patterns. The following demonstrates the above Quickdraw routines and the patterns available via PENPAT:

```
(DEFUN QukDrw ()
 (paintrect '(-52 -56 -32 -32))
 (penpat dkgray)
 (paintoval '(-32 -32 -8 -12))
 (penpat gray)
 (paintrect '(-8 -8 12 16))
 (penpat ltgray)
 (paintoval '(12 16 32 40))
 (penpat black)
 (paintrect '(36 44 52 64))
 (penpat white)
 (paintrect '(38 46 50 62)))
```

```
(QukDrw)
```

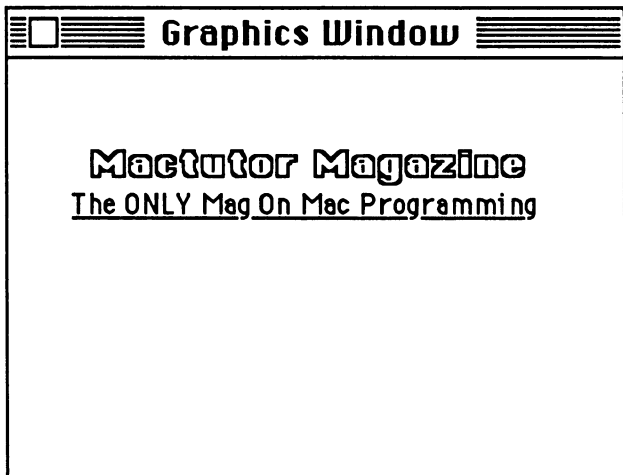


The first rectangle uses the default pattern, black. After each object is drawn the pen pattern is changed. The bottom rectangle is actually two. The first is drawn in black, the second is slightly smaller and drawn in white. If drawn by itself the last rectangle would not be visible.

DRAWSTRING puts text into the graphics window. TEXTFONT and TEXTFACE allow the manipulation of the font and style of the text produced from DRAWSTRING, respectively. TEXTSIZE changes the size of the text produced by DRAWSTRING. The following illustrates the syntax of these text control routines in ExperLisp:

```
(penup) (moveto -82 -35) (pendown)
(textsize 12) (textface (+ 1 8))
(drawstring "Mactutor Magazine")
(penup) (moveto -89 -20) (pendown)
(textfont 1) (textsize 10) (textface (+ 0 4))
(drawstring "The ONLY Mag On Mac Programming")
```

Penup and Pendown control when pen movement will and will not produce graphic output. Moveto moves the pen to the given coordinates. The above lists produce the following when compiled (see next page):



The text styles are controlled in TEXTFACE using numbers which represent the different styles. The styles and their respective numbers are as follows:

|                    |                      |
|--------------------|----------------------|
| <b>Bold</b> (1)    | <b>Shadow</b> (16)   |
| <i>Italic</i> (2)  | <u>Underline</u> (4) |
| <b>Outline</b> (3) | <b>Normal</b> (0)    |

Text fonts are also represented by numerals. The system font (Chicago) is represented by the number zero. Geneva is 1 and Monaco is 4. Other fonts are supposedly available, but it is not clear how the numbers are assigned. It is likely that the numbers representing the fonts are related to the font's resource ID. This is assigned by the DA/Font Mover just released by Apple.

Next month the Lisp Listener will continue with a discussion on iteration, more on Quickdraw routines and how Mouse input is achieved.



# Lisp Listener

## Iteration Techniques in Lisp



Andy Cohen  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 10

Last month I showed how ExperLisp uses some of the simpler Quickdraw routines. For the curious, the following table contains all the Quickdraw routines discussed in the ExperLisp Reference Guide.

### Quickdraw Routines

|                |                 |
|----------------|-----------------|
| CLOSEPOLY      | CLOSERGN        |
| COPYRGN        | DIFFRGN         |
| DISPOSERGN     | DRAWCHAR        |
| DRAWSTRING     | EMPTYRGN        |
| EQUALRGN       | ERASEARC        |
| ERASEOVAL      | ERASEPOLY       |
| ERASERECT      | ERASERGN        |
| ERASEROUNRECT  | FILLARC         |
| FILLOVAL       | FILLRECT        |
| FILLRGN        | FILLROUNDRECT   |
| FRAMEARC       | FRAMEOVAL       |
| FRAMEPOLY      | FRAMERECT       |
| FRAMERGN       | FRAMEROUNRECT   |
| GETFONTINFO    | HIDECURSOR      |
| HIDEPEN        | INSETRGN        |
| INVERTARC      | INVERTOVAL      |
| INVERTPOLY     | INVERTRECT      |
| INVERTRGN      | INVERTROUNDRECT |
| KILLPOLY       | LINE            |
| LINETO         | MOVE            |
| MOVETO         | NEWRGN          |
| OBSCURECURSOR  | OFFSETPOLY      |
| OFFSETRGN      | PAINTARC        |
| PAINTOVAL      | PAINTPOLY       |
| PAINTRECT      | PAINTRGN        |
| PAINTROUNDRECT | PENMODE         |
| PENNORMAL      | PENPAT          |
| PENPOS         | PENSIZE         |
| SECTRGN        | SETEMPYRGN      |
| SETRECTRGN     | SHOWCURSOR      |
| SHOWPEN        | SPACEEXTRA      |
| TEXTFACE       | TEXTFONT        |
| TEXTMODE       | TEXTSIZE        |
| UNIONRGN       | XORRGN          |

Quickdraw commands will be discussed in more detail as each command is used within the ExperLisp examples.

### Iteration

Iteration is a process which provides a method of performing repetitive actions. In BASIC, iteration is accomplished with the FOR...NEXT or GOTO statements. Fortran uses the DO loop and Pascal REPEAT.....UNTIL. Experlisp has a number of different kinds of iteration, some of

which look and act like those of the above.

The first kind of iteration requires use of the PROG special form. PROG stands for program and has a list of variables associated with it that are equal to nil when the PROG is evaluated. These variables are local and are bound to the PROG. The body of the PROG follows the variable list. The body contains forms or lists which are evaluated and consist of symbols, values or procedures. PROG by itself returns nil and it does not provide iteration. The special form GO within PROG is used with an arbitrary tag for iteration. The tag is placed where the loop begins within the PROG. GO *tagname*, placed further along within the PROG, puts the next evaluation to where the tag is. For example;

```
(defun iter () ;1
 (prog ((y 1)) ;2
 loop ;3
 (setq y (add1 y)) ;4
 (print y) ;5
 (if (not(= 6 y)) (go loop)))) ;6
```

(iter)

When the above is compiled...

```
;iter
;2
;3
;4
;5
;6
;nil
```

The semicolon on each line of the sample is the method one uses for placing remarks within ExperLisp source code. Everything after the semicolon is ignored. The above PROG is within a defined procedure (iter). This makes it possible for the code to be compiled and then called as a procedure. Multiple PROGS are allowed within a DEFUN. However, the variables are local to each of the PROGS. It is wise to keep the number of PROGS within a defined procedure down to a minimum. The smaller the procedures, the easier to debug. Line 2 starts the PROG, declares the variable "y" and gives "y" the value of 1. ExperLisp syntax requires the variable or variables and their associated values be contained within an overall variable list. That is why the list "(y 1)" is contained within parentheses. Line 3 is the GO tag. Note, that it is not within its own list. However, it is within the overall PROG list. In line 4 the value of y has the number 1 added to it. The value of this sum is then assigned to the symbol "y" using SETQ. Line 5 prints the value of "y" for each loop. Line 6 tests to see if the value of "y" equals ten. If it doesn't, then the

evaluation is sent to the GO tag, "loop". The entire process is then repeated until "(not(= 10 y))" returns "nil". If it does then the next evaluation takes place after line 6. Since Line 6 is actually the end of the PROG, nil is returned and the iteration is stopped. In order to get a value returned from a PROG one must use RETURN. RETURN not only returns a value, it also terminates the iteration. For example:

```
(defun iter (x)
 (prog ((y 1))
 loop
 (setq y (add1 y))
 (if(= x y) (RETURN "Y IS EQUAL TO X")
 (print y))
 (go loop)))

(iter 5)
```

In the above "x" is compared to "y". When found to be equal ExperLisp stops the iteration and returns the string as follows;

```
;iter
;2
;3
;4
;Y IS EQUAL TO X
```

Early versions of Lisp use PROG and GO for iteration. Later versions, which strive to conform to the Common Lisp standard, are using a more efficient form of iteration form called DO. DO not only specifies the variable label, it can indicate an initial value, increment the value and contain a conditional for halting the iteration. The syntax for the DO special form is as follows:

```
(DO ((Variable initial value
 increment) (Conditional))
 (form)
 (form)
 )

(Defun Iter2 (x)
 (DO ((y 1 (add1 y)))
 ((= y x) "Y IS EQUAL TO X")
 (Print y)))
```

The above is a DO version of the PROG sample, "Iter". It performs the same task however, it's output is slightly different.

```
; (iter2 5)
;1
;2
;3
;4
;"Y IS EQUAL TO X"
```

In the PROG version, the addition of the number one to

"y" is performed prior to the print command. Hence, the first number printed is two. In the DO version the ADD1 procedure is not performed until the second loop. The print "y" statement is performed in the first loop so that the number one is printed first. Note that when the conditional "(= y x)" is true the iteration is halted and what follows the conditional is returned. For another example, check out following:

```
(defun iterfun (x) ;1
 (do ((a 0 (add1 a))) ;2
 ((= a 5) 'done) ;3
 (print (* a x)))) ;4

(iterfun 3)

;iterfun
;0
;3
;6
;9
;12
;done
```

Line one of the above specifies the defined procedure and creates the symbol "x". In line two the variable "a" is initialized and assigned the value zero. On that same line the method of incrementation of the variable is also specified. Line three is the conditional and tests the variable "a" for equivalence with the number five. Line four prints the product of the value of "a" for each iteration and the value assigned to the symbol, "x". One of the features of DO is that one may use GOs and RETURNS just as in the PROG.

There is another form of DO for faster and simpler iteration. DOLIST assigns the elements of a list to a symbol. It then performs the functions within its body upon each element of the list one at a time. Its syntax is as follows:

```
DOLIST (Symbol '(list)) (functions.....)

(DOLIST (FUN '(Buy Mactutor Mag))
 (print fun))

;Buy
;Mactutor
;Mag
;nil
```

The print command is performed upon each member of the list assigned to "FUN". Note that it always returns nil when the iteration is complete. This is because the list assigned to "FUN" is exhausted. The list then contains nil.

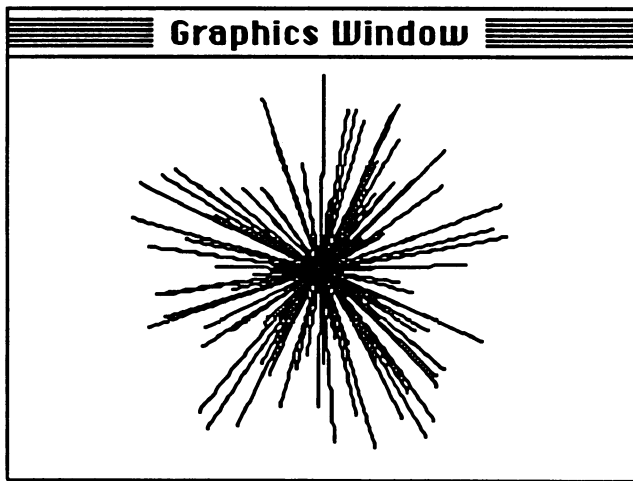
One of the easiest to use forms of iteration (especially to someone who knows BASIC) is DOTIMES. Using DOTIMES one specifies only a variable which represents each loop and the maximum number of loops one wants performed. For example (see next page):



```
(Dotimes (turn 150)
 (forward (/ turn 2))
 (back (/ turn 2))
 (right turn))
```

When compiled or typed into the Listener Window the above list produces the following:

The symbol "turn" is initially zero. The body of DOTIMES is iterated 150 times. I used 150 so that enough lines would be drawn to make it interesting. Remember FORWARD? Well, BACK moves the pen backward. I divided "TURN" in half when the pen is moved so that the lines don't go off the column's margins. I'll continue with another form of iteration after the next couple of paragraphs.



Using the Mouse

Using the mouse as an input or interactive device with ExperLisp is quite simple. GETMOUSE reports the X-Y coordinates of the mouse's pointer on the active window. Type the following in the Listener Window:

```
(GETMOUSE)
;(146 107)
```

BUTTON is a predicate which returns "t" when it is evaluated and the mouse's button is held down. It returns nil when the button is not held down. The following is an example of GETMOUSE and BUTTON in iterative functions.

```
(defun Watch ()
 (prog ()
 look
 (if (button) (Mousey)
 (go look))))
```

```
(defun Mousey ()
 (prog ()
 another
```

```
(print (getmouse))
(if (not(button))
 (halt) (go another))))
```

```
(defun halt ()
 (print "that's all folks!"))
```

```
(Watch)
```

The first of the three above procedures, Watch, waits for the mouse's button to be pressed. If it is not then the loop continues. When the button is pressed then Watch calls Mousey. Mousey will print the X-Y position of the mouse as long as the button is held down (remember, "(IF (then) (else))" ?). When the button is released Mousey calls Halt. Halt simply prints a message with no iteration.

One more form of iteration is the function, WHILE. WHILE includes a conditional. If the conditional returns true then the body of the WHILE is performed. What makes WHILE an iterative function and one that is different from those above, is that the conditional repeats the test as long as it returns true or nonnil. For example:

```
(While (Not(Button)) (Print(Getmouse)))
```

The above performs a lot like "Mousey" above. However it all takes place on one line and operates from the mouse differently. Instead of showing the mouse's position with the button down, the above shows until the button is pressed down.

One aspect of ExperLisp which might not be apparent to a novice is the fact that once a procedure is defined via DEFUN, it is available until one quits ExperLisp. ExperLisp is not just a language, it is an environment. If one was to compile all of the samples in this month's issue, then each sample can be run by simply typing the procedure's name within a list in the Listener Window. The value of this fact will be more apparent after the discussion on menus in ExperLisp.

## User Warning

In using ExperLisp one must be aware from the very beginning of a bug in ExperLisp's design of the user interface. When one tells the Mac to save from the menu selection under "File", ExperLisp saves the active window into the active file. If one has the Listener Window as the active window and tells ExperLisp to save, the entire file will be replaced by the contents of the Listener Window. BE CAREFUL. Check which window is active before you save. Eventually, later versions of ExperLisp will have a "snapshot" feature. This feature will save the contents of the environment. One will be able to get ExperLisp back to exactly where it was when the snapshot was taken without recompiling files. This might be performed by saving with the Listener Window active. If one has a couple of files compiled and is using them in

conjunction, this feature will save lots of time when saving and restarting ExperLisp.

Last month I described how one can measure a Lisp machine's capacity by seeing how many cons cells it can handle. I also described the FREECONS procedure available in the latest versions of ExperLisp (v1.02). I'd like to report that after upgrading my 512K Mac to 2 megabytes (from Levco in San Diego) I found that ExperLisp can have as many as 30,440 cons cells. However, that many cons cells is far from necessary for the simple examples contained in the Lisp Listener column. A 512K Mac will do fine. In writing this article though, I put Experlisp, MacWrite and MacPaint all into the Switcher utility (v3.5). Most of which I load from a RAMdisk! No more waiting to boot up the applications,

verify code, or copy and paste. I have Switcher allocate 500K to ExperLisp and I can still access 5000 cons cells. When using the full two megabytes, one has as many cons cells as a lot of the very expensive Lisp workstations. Note that the upgrades of 1 or 2 megabytes available by most companies do not necessarily access the extra memory in the same way. I can assure all those interested that the 2 megabyte (or Monster Mac) upgrade from Levco can access the extra RAM using ExperLisp or most of the other applications which were designed in accordance with the Macintosh guidelines.

Next month I will show how to use the Mouse in interacting with the Quickdraw routines. I will also discuss recursion.



# Lisp Listener

## Recursion and Windows



Andy Cohen  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 11

### MacScheme Offers An Alternative

This month I am happy to announce that there is now a third Lisp environment available for the Mac. The first was XLisp (which is now in version 5, which included some quickdraw routines), the second was ExperLisp (see below for the current version number) and the third is called MacScheme from Semantic Micro Systems in Beaverton, Oregon. MacScheme is an implementation of Scheme, which is actually an idealized version of Common Lisp, which was proposed by Guy Steele. MacScheme is implemented on the Macintosh in a different way than ExperLisp or XLisp. I will give the details of how it was developed in a more detailed review in the near future. For now though I can say that it will have the capability to manipulate over 20,000 cons cells with just a 512K Mac. I'm also told that a Levco MonsterMac (2 mbyte) using MacScheme can have approximately 120,000 cons cells. In contrast, ExperLisp can only create 5,912 cons cells in a 512K Mac and 30,480 in a MonsterMac. MacScheme however, will not have the bells and whistles that ExperLisp has. There is currently no way in MacScheme to access the Mac's rom toolbox. John Ulrich, one of the developers of MacScheme tells me that in time they eventually will be able to access the toolbox. MacScheme does use the standard Macintosh user-interface. It uses the menu bar and can produce up to four windows. These windows

contain things like a compiler, an editor, a debugger (!) and a Listener window. It is supposedly as interactive as ExperLisp. One can compile and evaluate a single line from the Listener window or compile an entire file. There are two more aspects to MacScheme which I feel are almost as significant as its capacity; cost and copy protection. The cost is extremely low, \$125, and it is not copy protected. It can easily be put onto a hard disk or a RAM disk for more speed during startup. As soon as I get MacScheme I will give a more detailed review. I hope to include some benchmarks and a better comparison to ExperLisp.

### Simple Recursion

Recursion is one of the most basic control structures in Lisp. To recur is to occur again. A recursive structure or function is one which is essentially iterative, in that it is repetitive. Recursive structures or functions are different from iterative procedures in that they call themselves repeatedly in a circular fashion in order to solve a problem. The recursion ends when the solution is computed. For example, suppose one needed to know how fast a printer can print a certain number of pages. Let us assume that we already know it takes .763 minutes per page. We have also just learned that a certain

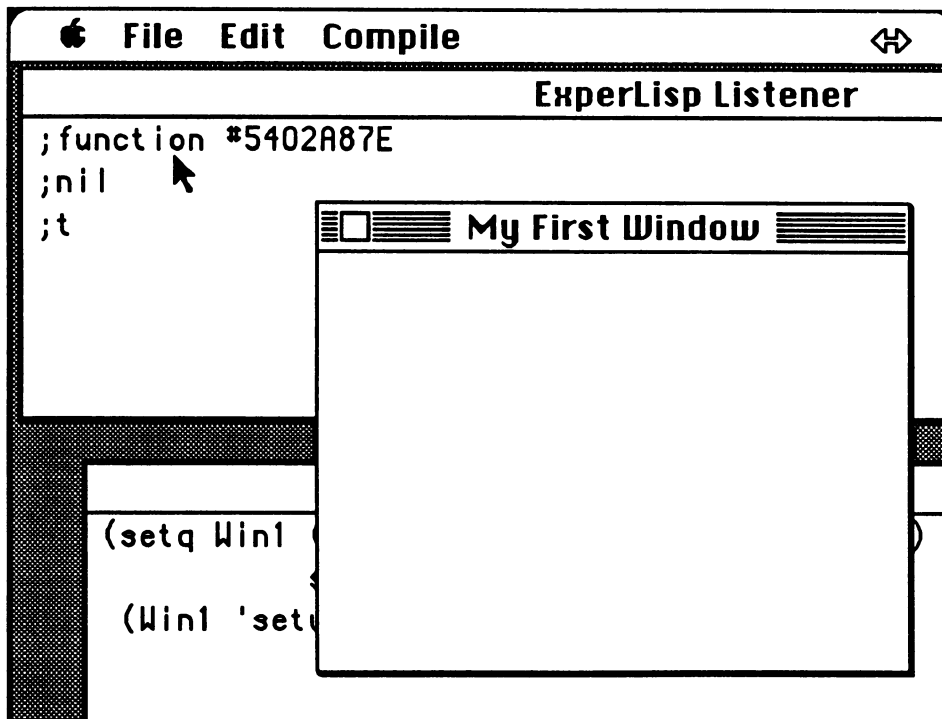
Master's thesis is 72 pages long. Well we could simply multiply .763 by 72, but that would not be an example of recursion. Multiplication would be the easy way. Let's try the hard way. What the following will do is add the time per page (.763) to a variable, which I have called timecount, once for each page.

```
(setq timecount 0)
(defun Printtime (pagenum)
 (setq timecount (+ timecount .763))
```

```
(cond ((= 1 pagenum) timecount)
 (t (Printtime (sub1 pagenum)))))
```

```
(Printtime 72)
;54.936
```

When the above is compiled, "timecount" is assigned the value zero. When "Printtime" is called the value sent with the call is the number of pages (pagenum). The second line of



Printtime reassigns to "timecount" the value of the sum of the previous value of timecount and the number .763. The next line is a conditional which tests the value of pagenum. If pagenum is one, the "=" predicate returns "t", the value of timecount is displayed and the procedure is terminated. If pagenum is not equal to the number one then the equal predicate returns "nil" and the next line is evaluated. The third line forces its evaluation by virtue of the "t" at the beginning of the list. This list then calls the very same defined function, however the parameter "pagenum" is decremented by one. This continues until pagenum equals zero. The time it took the printer to print the given number of pages is then displayed. In the case of 72 pages it took 54.936 minutes (it was probably high quality!)

Sometimes it requires more than one function to solve a problem. Two-part recursion is when one function calls a second and the second function does all the work. Since we actually need to have the variable "timecount" set to zero every time we use the function it can be more efficiently implemented in terms of our Lisp environment as a two-part recursive function.

```
(defun setup (n)
 (setq timecount 0)
 (Printtime n))

(defun Printtime (pagenum)
 (setq timecount (+ timecount .763))
 (cond ((zerop pagenum) timecount)
 (t (Printtime (sub1 pagenum)))))

(Setup 72)
; 54.936
(Setup 90)
; 68.67
(Setup 12)
; 9.156
(Setup 32)
; 24.416
```

Given the above, one can now solve the problem without having to compile the entire source listing every time. Using "setup" which calls "Printtime" the variable timecount is initiated without recompiling. Note that the value represented by variable "n" is passed to Printtime which then assigns it to "pagenum". If one was to call Printtime after compiling and running the above, the last number assigned to timecount will still be assigned. Therefore, the number returned is the total of the last total time plus the number of pages given when Printtime was called by itself. Recursion is an important feature of Lisp. The above was a very simple example. In next month's column, I'll discuss more complicated recursive functions.

## Windows!

Yes, ExperLisp does do windows. After months of saying that I'd show how, I've finally gotten around to doing it. As you might already know from previous months, one may generate a window by simply accessing either the Bunny or Quickdraw graphics routines available in ExperLisp. In this case the window is the standard graphics window or "Std\_graf" to ExperLisp. "Std\_graf" is the term used by ExperLisp in referring to a window. One may associate a "Std\_graf" with their own window by assigning the special term "newgraphwindow" and a list containing a set of coordinates (which correspond to the top left and bottom right corners of the window) to the chosen term which identifies the window. For my example the term I have chosen is "Win1". After this assignment is made one must then assign the name "Win1" to the special term "Std\_graf". To assign a title to the window which will be displayed on top of the window the special term "setwtile" is used. My sample follows:

```
(setq Win1 (newgraphwindow '(90 115 290 335))
 std_graf Win1))
(Win1 'setwtile "My First Window")
```

The above will generate a window which does not contain scroll bars or an expand corner. Expanding the window or shrinking it however, is possible by placing the mouse on the lower right corner and dragging it to the desired size. One might note that after running the above, the window is covered up by the Listener Window. It has to be selected before it can be changed or moved. This is because when there is no other functions operating the Listener Window becomes the active window. In order to keep the programmed window active one must be within a procedure which selected it or a procedure which is called by a previous procedure which selected the window. Figure 1 is a screen dump of what the window looks like on the ExperLisp desktop. The screen dump also contains the output generated within the Listener window as a result of the window creation. Windows can also be selected via quickdraw routines. For example:

```
(FILLOVAL '(34 67 89 90) dkgray Win1)
```

The window is selected by including the window title at the end of the quickdraw routine. The window can be deleted by using the following:

```
(CLOSEWINDOW)
```

One can size a window using the following:

```
(SIZEWINDOW width height)
```

One can also HIDEWINDOW, SHOWWINDOW, MOVEWINDOW *x y*, or SENDBEHIND *window*. All of the window primitives are as easy to use as those sampled above.

## Putting It All Together

Now let's put all that has been discussed over the last couple of months together into a simple bunch of procedures which will allow the user to draw a black rectangle with the mouse into the window generated by the code described above.

```
(defun Watch ()
 (Win1 'selectwindow)
 (prog ()
 look
 (if (button) (Mousey (REVERSE (getmouse))))
 (go look)))

(defun Mousey (x)
 (prog ()
 try
 (setq inputs (append x (REVERSE(getmouse))))
 (framerect inputs)
 (eraserect inputs)
 (if (not(button)) (halt x) (go try)))

(defun halt (x)
 (paintrect (append x (REVERSE (getmouse)))))

(Watch)
```

The overall structure of the above is the same as that used with the procedures presented last month, which printed the mouse location. When "Watch" is called, the window "Win1" is selected and brought to the front. The procedure "Watch" also scans the mouse awaiting a button press. When the button on the mouse is pressed the X-Y values of the mouse position are reversed (to correspond to the top-left of the quickdraw syntax) and sent in a list as a parameter to the procedure "Mousey". Mousey appends together a list made up of the passed parameter "x" and a second (REVERSE (GETMOUSE)). This list is assigned to the symbol "inputs", which is then used in the FRAMERECT. The FRAMERECT is immediately erased using ERASERECT to give the outline effect prior to releasing the button and selecting the rectangle size. When the button is released the procedure "Halt" is called with the exact same value passed to it as the value passed to "Mousey". It is still the top, left hand corner of our rectangle. In "Halt" the rectangle is finally drawn using the latest mouse position for the bottom, right hand corner.

When compiling these routines one should see each function name printed in the Listener Window as the function is compiled. If the name does not appear and the "I" beam cursor reappears, there is a typo. Chances are it is a problem with the parentheses. If the names show up but the functions are not initiated, chances are a parenthesis was left out somewhere. The above should work fine. One of the biggest disadvantages in using ExperLisp at this time is the poor error messages given when the programmer has done something wrong. I find that most of my errors are in leaving out a parenthesis or putting too many in. Parentheses matching is a capability inherent in an editor which gives some indication to

the programmer as to which parenthesis goes with another. Some Lisp machines actually put the parenthesis in for the programmer automatically. In version 1.04 of ExperLisp we will be given the luxury of parenthesis matching. Version 1.04 should be released by the time this column is in print. This new version of ExperLisp will handle parenthesis matching by highlighting the opening parenthesis of the matching closing parenthesis to the left of the entry point. By placing the entry point next to each closing parenthesis the programmer can check to see that all the lists are closed properly. Believe me, after just two minutes of use you will never want to be without parenthesis matching again! Another helpful feature of version 1.04 is a form of trace capability. In the beta version this capability was initiated within the \*lispINIT file. When certain kinds of errors are encountered the same kind of message is generated as was in the earlier versions, however with this form of trace on the message is followed by each list that was invoked and the order of occurrence. These are generated in the Listener window. There is a drawback to this feature though, the listing can get quite long for even a small program. The best way to debug at this time is still to break the code down into small functions.

## ExperLisp Versions

There are currently four versions of ExperLisp. Version 1.01 was the first release. The first update was version 1.02, in which some of the routines were fixed (i.e. MAPCAR and nested CARs and CDRs, etc.) and some routines added (i.e. FREECONS). Version 1.03 is a version of ExperLisp which was included with ExperOps5. Registered owners of both ExperLisp and ExperOps5 should receive version 1.04. There are many differences between each of these versions. To see what routines and functions are available within a version of ExperLisp type the following in the Listener window:

(APPROPOS "")

The function names will then be generated within the Listener window. Good luck, the list is long.

## The Experts Complain

Recently a letter was received by MacTutor which criticized the accuracy and reliability of this column. I would like to say that I am NOT a Lisp programmer. That is probably obviously apparent to one who is. The goal of this article is to be more than just a tutorial, it is to demonstrate the findings of one who is learning Lisp ON THE MAC with only minor consulting from the experts. If you feel something is incorrect, then by all means write to MacTutor with your comments. We welcome suggestions for topics of interest to our readership. Let us know how we can meet your needs.



# Lisp Listener

## Windows and Tic-Tac-Toe!



Andy Cohen  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 12

This month, the Lisp Listener will feature a program written by Dean Ritz of ExperTelligence. The program is a Tic-Tac-Toe game which uses all of the functions described in previous issues. However, before discussing the game I'd like to present a subject long overdue.

### Menus In ExperLisp

Creating a menu on the menu bar is actually a pretty straightforward, though nonintuitive, operation. In creating a menu one must use one of the special "hooks" in ExperLisp. There are also hooks for printing. Those will eventually be covered in future installments. The first step in creating a menu must be the identification of what the menu items will do. Menu items can call functions or perform assignment. If the items call defined functions, these functions should already be defined and the function names known. One then defines a function which returns a menu and a menu item on the condition that a certain menu and item position is selected. One easy way to do this uses a conditional which has not yet been discussed.

SELECTQ acts like COND since it has a clause and returns something based on the value of the clause. The first item following the term SELECTQ is the *key-form*. Each of the test items which follow are evaluated and the one which is the equivalent of the key-form causes the return of its corresponding value or list. For example;

```
(defun what (x)
 (selectq x
 (1 'one)
 (2 'two)
 (t 'all)))
```

```
(what 1)
one
(what 3)
all
```

In the above, the selectq evaluates the passed value (x) for equality with the first atoms of each of the three following lists. If the atom is equal to the passed value, the atom following the first is returned. As in COND the "t" forces the return of the atom "all". In assigning menu item positions one may have a SELECTQ within a SELECTQ. The first level SELECTQ finds the menu, the second finds the menu item. For example:

```
(defun menuselect (themenu theitem)
```

```
(selectq themenu
 (10 (selectq theitem
 (1 (function1B))
 (2 (function2B))
 (3 (function3B))))
 (11 (selectq theitem
 (1 (function1A))
 (2 (function2A))
 (3 (function3A))))))
```

The first SELECTQ locates which group of items belong to the menu number assigned to "themenu". The second SELECTQ locates the item belonging to the item number assigned to "theitem". After one generates the above defined function one must assign the values returned by the function to the special menu-hook;

```
(setq *menuhook menuselect)
```

One then assigns a label to the "NEWMENU" function (It is a built-in function) and its elements. This must be done with the ID number that one wants to assign to the menu as well as the menu title. For example:

```
(setq mymenu (newmenu 10 "menu B"))
```

Using this label, one then assigns the item labels to the menu as follows:

```
(appendmenu mymenu "function1B")
(appendmenu mymenu "function2B")
(appendmenu mymenu "function3B")
```

or one may do it all in one list as follows:

```
(appendmenu mymenu "function1B; function2B;
function3B")
```

One then inserts the menu with:

```
(insertmenu mymenu 0)
```

Then draws it with:

```
(drawmenubar)
```

The entire sample with dummy functions follows:

```
(defun function1A ()
 (print "function 1A"))
```

```
(defun function2A ()
 (print "function 2A"))
```

```
(defun function3A ()
 (print "function 3A"))
```

```
(defun function1B ()
 (print "function 1B"))
```

```
(defun function2B ()
 (print "function 2B"))
```

```
(defun function3B ()
 (print "function 3B"))
```

```
(defun menuselect (themenu theitem)
 (selectq themenu
 (10 (selectq theitem
 (1 (function1B))
 (2 (function2B))
 (3 (function3B))))
 (11 (selectq theitem
 (1 (function1A))
 (2 (function2A))
 (3 (function3A))))))
```

```
(setq *menuhook menuselect)
```

```
(setq mymenu (newmenu 10 "menu B")
 grmenu (newmenu 11 "menu A"))
```

```
(appendmenu grmenu
 «function1A;function2A;function3A»)
(appendmenu mymenu
 "function1B;function2B;function3B")
```

```
(insertmenu grmenu 0) ;0=the beforeID #
(insertmenu mymenu 0)
```

```
(drawmenubar)
```

Note that in the INSERTMENU lists I used "0" as the *before ID* number. The number zero indicates that I want to place the menu to the right of any already existing menus.

One may call any of the six defined functions by selecting a menu item rather than by typing the function name into the Listener Window. We can see now that an application in ExperLisp can consist of a large number of functions which do not necessarily call each other. A function, when called, can provide some service on data or graphics as a stand alone utility. The operator can then change the data or graphics by calling up another function from the menu.

Another menu example follows:

```
(Defun number (x)
```

```
(cond ((= 1 x) '(the number is one))
 ((= 2 x) '(the number is two))
 ((= 3 x) '(the number is three))
 ((= 4 x) '(the number is four))
 ((= 5 x) '(the number is five))
 ((= 6 x) '(the number is six))
 ((= 7 x) '(the number is seven))
 ((= 8 x) '(the number is eight))
 ((= 9 x) '(the number is nine))
 ((= 10 x) '(the number is ten))))
```

```
(defun order (themenu theitem)
 (if (not (eq nil std_graf)) (std_graf 'selectwindow))
 (selectq themenu
 (10 (selectq theitem
 (1 (number 1))
 (2 (number 2))
 (3 (number 3))
 (4 (number 4))
 (5 (number 5))
 (6 (number 6))
 (7 (number 7))
 (8 (number 8))
 (9 (number 9))
 (10 (number 10))))))
```

```
(setq *menuhook order)
```

```
(setq mymenu (newmenu 10 "numbers"))
```

```
(appendmenu mymenu «one; two; three; four;
 four; five; six; seven; eight; nine; ten»)
```

```
(insertmenu mymenu 0)
```

```
(drawmenubar)
```

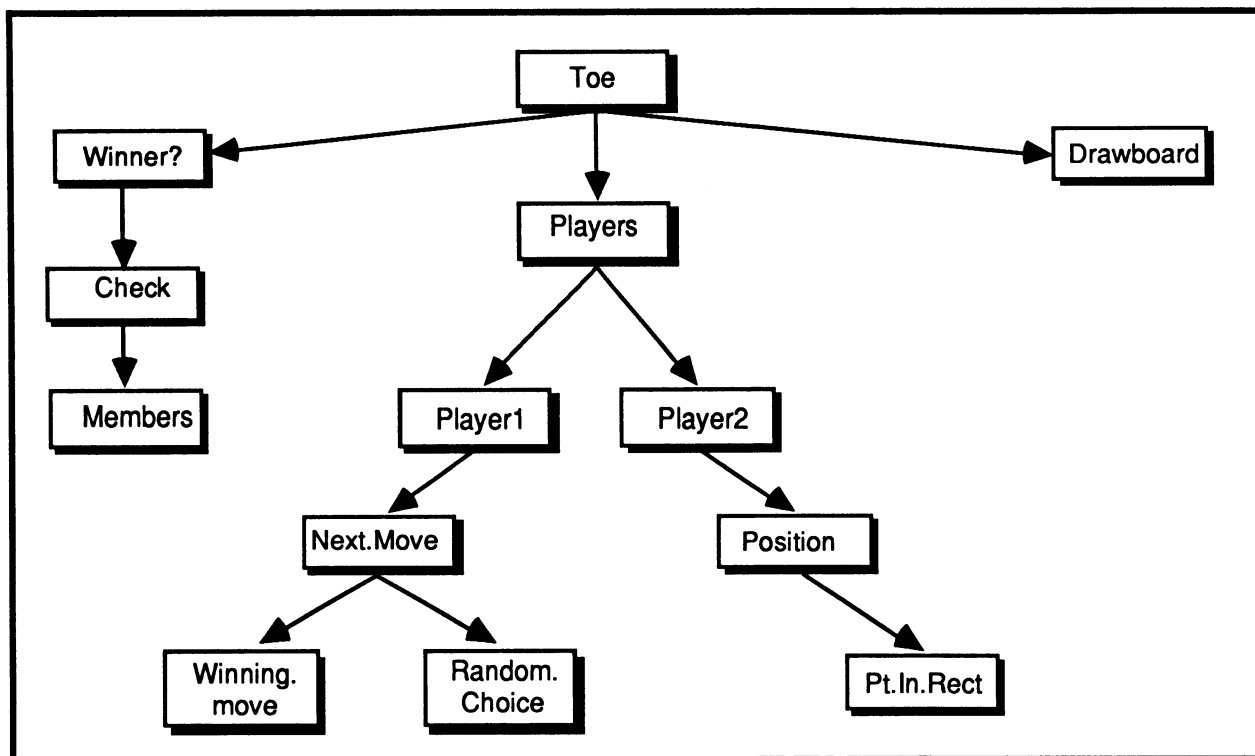
## Deleting A Menu

Deleting a menu is easily accomplished by using the built in DELETEMENU and DISPOSEMENU functions. DELETEMENU actually deletes the menu from the menu bar. DISPOSEMENU releases the memory used by the just deleted menu. One should then redraw the menu bar with DRAWMENUBAR.

```
(deletemenu 10)
(disposemenu mymenu)
(drawmenubar)
```

## Tic-Tac-Toe

While the following has very little that has not been discussed in previous Lisp Listener installments, there is a lot which is in different form. For example the use of SETQ for multiple assignment. ExperLisp accepts the spaces between the symbols and the stuff assigned to them as dividers.



However, these spaces are ignored by ExperLisp whether they consist of one space, multiple spaces, or one line. Therefore, one may make multiple assignment in a formatted manner for easier reading. See PLACES below.

In the function named "random.choice" MEMQ is used. MEMQ is similar to MEMBER. Where member tests to see if the given atom is a member of a list using EQUAL, MEMQ uses EQ. EQUAL test equality of characters regardless of whether they are upper case or lower case. EQ returns "t" only if the two items are virtually identical. MEMQ returns the same as MEMBER (that is, when the tested atom is contained in the tested list it is returned along with all the atoms which followed it in the list tested for the membership).

PUSH (used in defined function "Player2") is similar to APPEND. It does a CONS of the item following into a list.

Figure 1 is the functional breakdown of the Tic-Tac-Toe game. TOE calls WINNER? and PLAYERS which return values. TOE also calls DRAWBOARD which does what its title suggests. Both WINNER? and PLAYERS refer to subfunctions which use subsubfunctions. WINNER? calculates whether or not the game was either won or was a draw by looking at the number of moves made. If WINNER? returns nil then there is no winner yet and PLAYERS is called. PLAYERS checks to the current player's turn and calls either PLAYER1 or PLAYER2 based on which turn it is. PLAYER1 is the computer's moves. It uses NEXT.MOVE,

WINNIN.MOVE and RANDOM.CHOICE to generate the move. Player2 uses POSITION and PT.IN.RECT to allow the human player to enter a move using the mouse. To start the game enter "(TOE)" into the Listener Window.

```

;*****
;Tic-Tac-Toe written by Dean Ritz
;PLACES is a list of pairs. The first element of each
;list of pairs represents the position (1-9). The last
;second element in each list of pairs is a boundary
;list for the piece which fits into that position.
;TURN remembers whose turn it is (player 1 or 2).
;WINS is a list of the 8 possible winning sets. The
;program checks possible moves against this list.
;*****
(defun toe ()
 (setq
 p1 nil
 p2 nil ;Set the chosen positions to empty lists.
 places '((1 (-85 -85 -35 -35)) (2 (-85 -25 -35 25))
 (3 (-85 35 -35 85)) (4 (-25 -85 25 -35))
 (5 (-25 -25 25 25)) (6 (-25 35 25 85))
 (7 (35 -85 85 -35)) (8 (35 -25 85 25))
 (9 (35 35 85 85)))
 wins '((1 2 3) (4 5 6) (7 8 9) (1 4 7) (2 5 8)
 (3 6 9) (1 5 9) (3 5 7))
 turn 1
 std_graf (newgrafwindow '(45 5 320 500)))
 (send std_graf 'setwtitle "Tic Tac Toe")
 (send std_graf 'showwindow)
 (send std_graf 'selectwindow)

```



```

(textface 0)
(drawboard)
(while (not (winner?)) (players))
(cond ((check wins p2)
 (fillrect '(-110 -201 10 -100) white)
 (moveto -200 0)
 (drawstring "You win!!"))
 ((check wins p1)
 (fillrect '(-110 -201 10 -100) white)
 (moveto -200 0)
 (drawstring "The Macintosh wins!!"))
 (t
 (fillrect '(-110 -201 10 -100) white)
 (moveto -200 0)
 (drawstring "Cat's game.")))
(dotimes (i 1800) (add1 3))
(std_graf 'closewindow)
(setq std_graf nil)

;*****
;
(defun drawboard ()
 (pendown)
 (penpat gray)
 (framerect '(-90 -90 90 90))
 (framerect '(-90 -32 90 32))
 (framerect '(-32 -90 32 90))
 (penpat black))

;*****
;WINNER? uses brute force to see if there is a
; winner. If turn is 1, then it checks
; to see if player 2 won the game
; with the last move.

(defun winner? ()
 (cond ((< 8 (+ (length p1) (length p2))) t)
 ((and (= turn 1)
 (check wins p2)) 2)
 (t
 (and (check wins p1) 1))))

;*****
;
(defun check (for against)
 (cond ((null for) nil)
 ((members (car for) against) t)
 (t (check (cdr for) against))))

;*****
;MEMBERS works recursively to see if all of the
; elements of the list :L1
; are members of the list :L2.

(defun members (l1 l2)
 (cond ((null l1) t)
 ((memq (car l1) l2)
 (members (cdr l1) l2))))

;*****
;This toggles the variables which determines whose
; turn it is.

```

```

(defun players ()
 (cond ((= turn 1)
 (player1)
 (setq turn 2))
 (t
 (player2)
 (setq turn 1))))

;*****
;This runs the computer player's move.

(defun player1 (&aux choice)
 (fillrect '(-110 -201 10 -100) white)
 (moveto -200 0) (drawstring "Macintosh")
 (setq choice (next.move)
 p1 (cons choice p1))
 (filloval (car (last (assq choice places))) ltgray))

;*****
;this is the psuedo brains of the whole thing. The
; goal is first to select a move that wins, then
; select a move that blocks,
; otherwise just select any untaken shot.

(defun next.move ()
 (prog (tempo)
 (setq tempo (winning.move 1 p1))
 (if tempo (return tempo))
 (setq tempo (winning.move 1 p2))
 (if tempo (return tempo))
 (random.choice)))

;*****
;
(defun winning.move (count p)
 (prog ()
 begin
 (cond ((or (memq count p1) (memq count p2))
 (setq count (iadd count 1))
 (go begin))
 (> count 9) (return nil))
 ((check wins (cons count p)) (return count))
 (t (winning.move (iadd 1 count) p))))

;*****
;This returns a randomly chosen position that has not
; already been chosen.

(defun random.choice ()
 (prog (temp)
 loop
 (setq temp (iadd 1 (random 9)))
 (if (or (memq temp p1) (memq temp p2))
 (go loop) ;If that position is chosen, get another.
 temp))) ;Otherwise output the position.

;*****
;This manages the human player's move.

(defun player2 ()
 (prog (choice)
 (fillrect '(-110 -201 10 -100) white)

```

```

(moveto -200 0)
(drawstring "You")
loop
(setq choice (position))
(if (or (memq choice p1) (memq choice p2))
 (go loop))
(push choice p2)
(filloval (car (last (assq choice places))) gray)))

```

\*\*\*\*\*

;This gets a position from the mouse.

```

(defun position ()
 (prog (pt count)
 begin
 (if (not (button)) (go begin))
 (setq pt (getmouse))
 (if (not (pt.in.rect
 (car pt) (car (last pt))
 '(-85 -85 85 85)))
 (go begin))

 (setq count 1) ;incremented for a position number
 pointer
 (if (pt.in.rect (car pt)

```

```

(car (last pt))
(car (last (nth (isub count 1) places))))))
(return count)
(progn
 (setq count (iadd count 1))
 (go pointer))))))

```

\*\*\*\*\*

;PT.IN.RECT tests to see whether a  
 ;specific X and Y coordinate lies within  
 ;a given boundary rectangle :RECT.  
 ;RECT should be a list of [TOP LEFT BOTTOM  
 ; RIGHT] coordinates.

```

(defun pt.in.rect (x y rect)
 (and (< x (nth 3 rect))
 (≥ x (nth 1 rect))
 (≤ y (nth 2 rect))
 (≥ y (nth 0 rect))
 t)) ;returns T if true, NIL otherwise

```

Next month will include an in-depth description of  
 MacScheme, the first installment of an ExperOps5 tutorial,  
 and a couple more Lisp functions.



# Lisp Listener

## 3-D Rotations



Andy Cohen  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 13

### Mapping Functions

Applicative operators are functions which use other functions as inputs. One of the most typical across Lisp dialects is MAPCAR. MAPCAR can perform an operation on each member of one given list, sequentially. It is therefore another form of iteration. MAPCAR operates in the same manner APPLY works in the following:

```
(apply + '(2 3 4 5))
14
```

MAPCAR, however, provides the capability to perform a given function for each atom within a given list. It can be seen as an "APPLY\_TO\_ALL". For example, suppose one wanted the square root of each value in a list of five values and create a corresponding list of these square roots. One way to do this is to remove the values from the list with nested CARS. Then, after applying the SQRT function to each value, a new list would need to be produced with a CONS. This method might get needlessly tedious. MAPCAR makes it possible to get this list in a much more convenient fashion.

```
(MAPCAR (lambda (x) (SQRT x)) '(2 3 4 5))
(1.41421356 1.73205080 2. 2.23606797)
```

LAMBDA is a special word that tells Lisp that what is following is to be treated as a function similar to DEFUN. LAMBDA acts like a one time only DEFUN. The (x) is the passed value and the list with the square root primitive is the expression to be carried out. MAPCAR takes the first value from the list (2 3 4 5) sequentially (that is why it is MAPCAR) and places it into x. It then puts each computed value into a list. How about another sample:

```
(setq x '(2 5)) (setq y '(5 7))
(mapcar (lambda (n) (* .5 n)) (append x y)))
(2 5)
(5 7)
(1. 2.5 2.5 3.5)
```

APPEND puts the values represented by the symbols x and y into a list. MAPCAR takes one atom at a time from the new list and multiplies it by .5.

MAPLIST is another form of mapping or sequencing function. However, instead of sequencing through a list one atom at a time MAPLIST performs a function on two entire lists. For example:

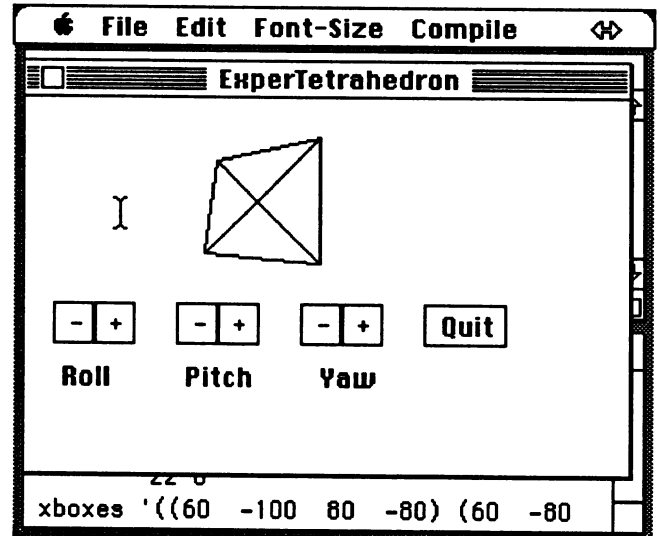


FIGURE 1. Tetrahedron Display

```
(MAPLIST APPEND '(A B C) '(D E F))
((A B C D E F)(B C E F)(C F))
```

MAPLIST removes the first atom of both lists and performs the function on the two new lists. It then removes the next two atoms and performs the function on the two lists with the remaining atoms. This continues until there are no more atoms in one of the two lists. The lists do not have to contain the same number of atoms.

```
(MAPLIST APPEND '(A B C) '(D E F G H))
((A B C D E F G H)(B C E F G H)
(C F G H))
```

MAPCAN and MAPCON are just like MAPCAR and MAPLIST, respectively except that they do not return lists that are made using LIST. MAPCAN and MAPCON use NCONC. NCONC takes the values from two lists and places all of them into the first, thereby destroying the original list. For example:

```
(SETQ x '(1 2 3 4) y '(5 6 7 8))
(NCONC x y)
(5 6 7 8)
```

```
(1 2 3 4 5 6 7 8)
x
(1 2 3 4 5 6 7 8)
y
(5 6 7 8)
```

NCONC put all eight values into "x" while it left "y" alone. Since it changed "x" it is considered destructive. An example using MAPCAN follows:

```
(MAPCAN (LAMBDA (x) (AND (NUMBERP x)
 (SETQ y (SQRT x)) (LIST y))) (2 3 4 5 A))
(1.41421356 1.73205080 2. 2.23606797)
```

The AND and the NUMBERP functions in the above, give indication as to when the numbers in the list end by having NUMBERP return nil from the letter "A". AND then stops evaluation. Otherwise the SETQ assigns the square root of each atom to "y" then places it within a list with LIST. Since MAPCAN was used each of the returned values were NCONCed into "y". If one used MAPCAR each value would be placed into a list represented by "y" which would then be placed into the resulting list. The SETQ changes "y" for each value and the resulting list is quite different.

```
(MAPCAR (LAMBDA (x) (AND (NUMBERP x)
 (SETQ y (SQRT x)) (LIST y))) (2 3 4 5 A))
((1.41421356)(1.73205080)(2.)
 (2.23606797) nil)
```

MAPC and MAPL are also related to MAPCAR and MAPLIST, respectively. These functions are supposed to return the original input list of values instead of a list of resultant values. They are typically used for their side effect such as assigning a new value to a global variable. Unfortunately they don't seem to return results in this manner. Instead of returning the original input list, they both return nil.

EVERY is a totally different type of sequencing function. EVERY applies a predicate to each atom in a list. If the predicate returns "t" for each atom EVERY returns "t". If the predicate returns nil at least once so does EVERY.

```
(EVERY NUMBERP '(1 2 3 4 5))
t
(EVERY NUMBERP '(1 2 A 4 5))
nil
```

One interesting feature about EVERY is that one may specify how EVERY will sequence through the list. Without specifying the sequence EVERY defaults to CDR as a step. For example, if we take the second EVERY example above and step through the list with a CDDDR, it returns "t" since the CDDDR makes the evaluation of the predicate skip the letter "A".

```
(EVERY NUMBERP '(1 2 A 4 5) cdddr)
```

t

FIND-IF applies a predicate function to each atom in a list until the predicate returns "t". FIND-IF returns the atom which satisfies the predicate then ends the evaluation.

```
(FIND-IF EVENP '(3 5 7 8 10))
8
(FIND-IF (lambda (x) (> x 5)) '(2 3 4 5 6 7 8))
6
```

FIND-IF-NOT does the exact opposite of FIND-IF. It returns the first atom of a list which does *not* satisfy the predicate.

```
(FIND-IF EVENP '(2 3 5 7 8))
3
(FIND-IF-NOT (lambda (x) (> x 5)) '(2 3 4 5 6))
2
```

REDUCE performs a function that requires two inputs. It performs the function on the first two atoms of a list, starting from left to right, then performs the function upon the result of the first two with the very next atom. It then performs the same function with the second result with the next atom. It returns the result when no more atoms are available. For example the following multiplies a sequence of numeric values:

```
(REDUCE * '(7 6 13 76))
41496
```

If one places a T before the last closing parenthesis REDUCE performs the same function on the list from right to left.

```
(REDUCE - '(7 6 13 76))
-88
```

The above is equal to  $((-(-7\ 6)13)76)$ .

```
(REDUCE - '(7 6 13 76) T)
-62
```

The above is now equal to  $(-7\ (-6\ (-13\ 76)))$ .

NOTANY is the equivalent of (NOT (EVERY.... It returns "t" if none of the atoms of a list satisfy the given predicate.

NOTEVERY is the same as FIND-IF-NOT. However, instead of returning the value that did not satisfy the predicate NOTEVERY returns "t" as follows:

```
(NOTEVERY (lambda (x) (> x 5)) '(2 3 4 5 6))
t
```

Besides MAPC and MAPL there is one more mapping function which not only doesn't work as the literature

(Common Lisp by Guy Steele) describes it, but just doesn't seem to work in ExperLisp version 1.04. This function is called SOME. SOME is supposed to look successively to the elements of a list and stop when it finds an element which satisfies a predicate function. The following should return the number five:

```
(defun try (x)
 (oddp x))

(SOME try '(2 4 5))
```

Instead of returning "5" after the defined function name, "unbound variable f" is returned along with a dump of what seems to be the heap. We will be talking to ExperTelligence about the status of these functions.

Last month I mentioned that we would be starting a tutorial on ExperOps5 in this month's issue. Unfortunately, the author of that segment did not make the deadline. We hope to start the tutorial as soon as we can, if it is possible, in coming Mactutor issues. Instead, we are again treated to a nifty program by Dean Ritz of ExperTelligence. The following functions build a small pyramid-like object in three dimensions. Three controls are also produced as seen in Figure 1. These controls rotate the object along three different planes and at varying speeds. After compiling, type (Tetrahedron) into the Listener window and enjoy!

By the way, Happy Holidays!

```
;*****
;TETRAHEDRON animates a tetrahedron rotating in ;3-
;Dimensional space.
;It allows a person to adjust the rate of rotation
;through the use of mouse sensitive controls.
;Produced by Dean Ritz of ExperTelligence
(defun tetrahedron (&aux (curbun (new3dbun)))
 (setq xx 0
 yy 0
 zz 0
 xboxes '((60 -100 80 -80) (60 -80 80 -60))
 yboxes '((60 -40 80 -20) (60 -20 80 0))
 zboxes '((60 20 80 40) (60 40 80 60))
 quit '(60 80 80 120)
 std_graf (newgrafwindow '(45 5 310 500)))
 (std_graf 'settitle "ExperTetrahedron")
 (std_graf 'showwindow)
 (std_graf 'selectwindow)
 (textface 0) (pendown)
 (draw.controls)
 (penup) (home) (forward 40)
 (pendown)
 (catch 'flag (doit 1))
 (disposhandle (coercetype $68 curbun))
 (std_graf 'closewindow))

(defmacro rac (l)
```

```
`(car (last ,l)))
```

```
;Hitting a key represent
(defun doit (tt &aux speed)
 (cond ((keyp)
 (setq speed (read-char))
 (if (numberp speed) (setq tt speed))))
 (if (button) (adjust.controls))
 (roll yy)
 (pitch xx)
 (yaw zz)
 (fillrect '(-130 -100 50 100) white)
 (dotimes (i tt) (tetra 70))
 (doit tt))
```

```
;*****
;DRAW.CONTROLS draws the mouse sensitive controls.
```

```
(defun draw.controls ()
 (moveto -100 75)
 (drawstring " - +")
 (framerect (car xboxes))
 (framerect (rac xboxes))
 (moveto -40 75)
 (drawstring " - +")
 (framerect (car yboxes))
 (framerect (rac yboxes))
 (moveto 20 75)
 (drawstring " - +")
 (framerect (car zboxes))
 (framerect (rac zboxes))
 (moveto 80 75)
 (drawstring " Quit")
 (framerect quit)
 (moveto -100 100)
 (drawstring " Roll Pitch Yaw"))
```

```
;*****
;ADJUST.CONTROLS is only called if the mouse button is
;depressed.
;It is responsible for calling the commands which adjust the
;rotation of the tetrahedron. It also sets the QUIT flag if
;the mouse is clicked in the "Quit" box.
(defun adjust.controls (&aux (point (getmouse)))
 (cond ((pt.in.rect (car point) (rac point) '(60 -100 80 -60))
 (apply adjust.x point))
 ((pt.in.rect (car point) (rac point) '(60 -40 80 0))
 (apply adjust.y point))
 ((pt.in.rect (car point) (rac point) '(60 20 80 60))
 (apply adjust.z point))
 ((pt.in.rect (car point) (rac point) quit)
 (invertrect quit)
 (wait) (invertrect quit)
 (throw 'flag))))
```

```
;*****
;The three commands ADJUST.X, ADJUST.Z, and ADJUST.Y
;are only
;called if the mouse is clicked while on one of the controls.
;It inverts the proper button (box), increments a global
;variable for moving the bunny, waits for the mouse button to
;be released, and then re-inverts the button.
```

```
(defun adjust.x (x y)
 (cond ((pt.in.rect x y (car xboxes))
 (invertrect (car xboxes))
 (setq xx (- xx 2))
 (wait)
 (invertrect (car xboxes)))
 (t
 (invertrect (rac xboxes))
 (setq xx (+ xx 2))
 (wait)
 (invertrect (rac xboxes))))))
```

```
(defun adjust.z (x y)
 (cond ((pt.in.rect x y (car zboxes))
 (invertrect (car zboxes))
 (setq zz (- zz 2))
 (wait)
 (invertrect (car zboxes)))
 (t
 (invertrect (rac zboxes))
 (setq zz (+ zz 2))
 (wait)
 (invertrect (rac zboxes))))))
```

```
(defun adjust.y (x y)
 (cond ((pt.in.rect x y (car yboxes))
 (invertrect (car yboxes))
 (setq yy (- yy 2))
 (wait)
 (invertrect (car yboxes)))
 (t
 (invertrect (rac yboxes))
 (setq yy (+ yy 2))
 (wait)
 (invertrect (rac yboxes))))))
```

```
;*****
```

;WAIT waits until the mouse button is depressed.

;Then it returns control to the calling function.

```
(defun wait ()
 (prog ()
 top
 (if (button)
 (go top))))
```

```
;*****
```

;TETRA and PART draw a tetrahedron using 3-D bunny

;graphics.

```
(defun tetra (s)
 (dotimes (i 3)
 (part s) (roll 90)
 (lt -45) (fd s)
 (bk s) (lt 45) (roll -90)))
```

```
(defun part (s)
 (lt 45) (fd s) (rt 45)
 (pitch 90)
 (rt 45) (fd s) (bk s) (lt 45)
 (roll -90))
```

```
;*****
```

;PT.IN.RECT tests to see whether an specific X and Y  
;coordinate  
;lies within a given boundary rectangle :RECT.  
;RECT should be a list of [TOP LEFT BOTTOM RIGHT]  
;coordinates.

```
(defun pt.in.rect (x y rect)
 (and (< x (nth 3 rect))
 (≥ x (nth 1 rect))
 (≤ y (nth 2 rect))
 (≥ y (nth 0 rect))
 t)) ;returns T if true, NIL otherwise
```



# Modula 2 Mods

## Modula's History



John Bogan  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 5

**Abstract:** MODULA Corporation of Provo, Utah has shipped a \$90.00 *Beta* site version of Modula-2 which runs reasonably well on a single disk 128K Macintosh. This paper attempts to make the case that Modula-2 deserves a thorough analysis to determine if it is the ideal standalone development language for serious commercial software on the Apple Macintosh. The conclusion of this paper is that Modula-2 will soon become the language of choice for farsighted software developers through the rest of the 1980's, and it is through the efforts of enlightened companies such as MODULA Corp. that this joyous result will come about.

### A Short History of Modula-2

The history of Modula-2 is the history of its designer, Dr. Niklaus Wirth. We pick up the story in the late 1960's when Dr. Wirth was a member of a large committee whose task was to design a successor to the language ALGOL-60, a highly popular language in Europe. ALGOL-60 is widely regarded as the inspiration for the discipline of **Structured Programming** which is the most significant breakthrough in software engineering in the past two decades. The result of this committee's efforts was ALGOL-68, a kludge. No coherent design philosophy guided this effort, resulting in a chaotic collection of features reminiscent of PL/I. In short, failure.

At this point Dr. Wirth abandoned the committee approach and decided to design his own language. Influenced by the structured programming theories of Dijkstra, Hoare and Dahl, Dr. Wirth set out to construct a language he could use to teach his own students the fundamentals of good programming. He began with some modest goals.

"The development of the language Pascal is based on two principal aims. The first is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language. The second is to develop implementations of this language which are both reliable and efficient on presently available computers." Pascal Report, p1, 1974, Niklaus Wirth

The rest is history. As anyone who follows the computer industry well knows, Pascal is one of the major influences on the software world today, particularly in the microcomputer world. To cite only one example, the second largest manufacturer of personal computers in 1984 appears to have adopted Pascal as its standard design language. That company is, of course, Apple Computer, Inc. As I understand it, most of the algorithms, data structures and techniques designed into the LISA and the Mac began life as Pascal programs.

But Pascal has significant flaws. Volumes have been

written disparaging Pascal as a systems programming language, and as a vehicle for writing applications of all stripes and colors. (Never mind that the language was never intended to do any of these things.) Further, and more disturbing, Pascal has been repeatedly bastardized by its implementors with a wide variety of (incompatible) extensions. After a decade in the field, there is no such thing as Standard Pascal any more than there is a tooth fairy. Nevertheless, many talented software people believe that even with its flaws, Pascal is still the best practical general purpose language we have at our disposal. I believe this situation is about to change.

On with Wirth's story. After largely washing his hands of Pascal Dr. Wirth turned his attention to multitasking and the problems of building large commercial programs, using teams of programmers. As it happens, Pascal is utterly unsuited for both of these activities. So Wirth designed himself a new, very simple language he called Modula. This language was never particularly practical it was more like a "hack" from one of the more gifted computer scientists around. However, it laid the groundwork for the concepts of separate compilation, modularity and libraries.

In 1976 Dr. Wirth took a sabbatical from his mountain hideaway, Eidgenössische Technische Hochschule, Institut für Informatik, better known as ETH in Zürich, Switzerland and visited that fount of inspiration - Xerox PARC. As Mac enthusiasts, we all know what that means: personal workstations, high resolution bit maps, mice, windows, icons, Smalltalk etc. Less well known is the Xerox system's implementation language, MESA. This language is a derivative of Pascal suitable for team implementation of low-level systems programs. The pieces were falling into place. As with many other visitors to PARC, Dr. Wirth came away inspired.

Dr. Wirth was tired of the bitching and moaning surrounding Pascal.

"If a language proves to be only marginally suitable for some application that was obviously not envisaged by its originator, we should muster the courage to build a new, truly adequate tool, instead of just grafting a fix onto the existing one." *Byte*, April 1983, pg. 386, Niklaus Wirth.

Dr. Wirth's inspiration was to build a new computer system from the ground up. A machine which would be programmed in one language from soup to nuts. From OS to text editor to graphics editor to who-knows-what, databases, spreadsheets, adventure games, whatever. Recognizing that a one language machine would require the use of dangerous low-level facilities as well as the safety, convenience and productivity of high level structured features, he sifted through his

encyclopedic knowledge of Pascal, Modula and MESA and in 1978 he defined his new language. To quote:

"Modula [1], a small language...featured a facility to partition programs into modules with explicitly specified interfaces. This was precisely the facility needed to allow the introduction of so-called low-level facilities in the high-level language, because it allowed you to encapsulate them and to constrain their dangerousness to clearly delineated parts of a program. Hence, the choice for the new language was Pascal, augmented by the module and a few other facilities, and regularized by a more systematic syntax. Thus was born Modula-2." *Byte*, August 1984, pg. 146, Niklaus Wirth.

Dr. Wirth had his ideal language. He then sat down with a couple of hardware types and some graduate students and built a bit-slice (AM2901) single-user high resolution graphic workstation with windows, mouse and multiple font editor with laser printer, long before the Macintosh ever saw the light of day! He called his machine *Lilith*. The sole American distributor for the Lilith and the Modula-2 compiler which this machine is built around is MODULA Corporation of Provo, Utah, also the makers of Modula for Mac.

### Modula Benchmark

The following program is an implementation of the Sieve of Eratosthenes expressed in Modula Corp's version of Modula-2 for the Apple 128K Mac. The following results were obtained when this program was compiled and executed.

The source (Prime.MOD) = 1,617 bytes.  
 Compile time = -1:45  
 The M-code (Prime.REL) = 324 bytes.  
 Link time = -1:20  
 The execution module = 4,348 bytes.  
 Execution time = -1:11 or 71 seconds.

Modula-2 Compiler, Interpreter Ver. 1.0.

```

MODULE Prime; (* Sieve of Eratos. *)
(* $R- disable subrange and type conv *)
(* $T- disable index checking *)
FROM InOut IMPORT WriteLn, WriteCard, WriteString;
FROM Terminal IMPORT ClearScreen;
CONST Size = 8190;
VAR i, prime, k, count, iter : CARDINAL;
 Flags : ARRAY[0..Size] OF BOOLEAN;
BEGIN
 ClearScreen;
 WriteLn; WriteString("10 iterations");
 FOR iter := 1 TO 10 DO
 count := 0;
 FOR i := 0 TO Size DO Flags[i] :=
 TRUE END;
 FOR i := 0 TO Size DO
 IF Flags[i] THEN
 prime := i + i + 3;
 k := i + prime;
 WHILE k <= Size DO
 Flags[k] := FALSE;

```

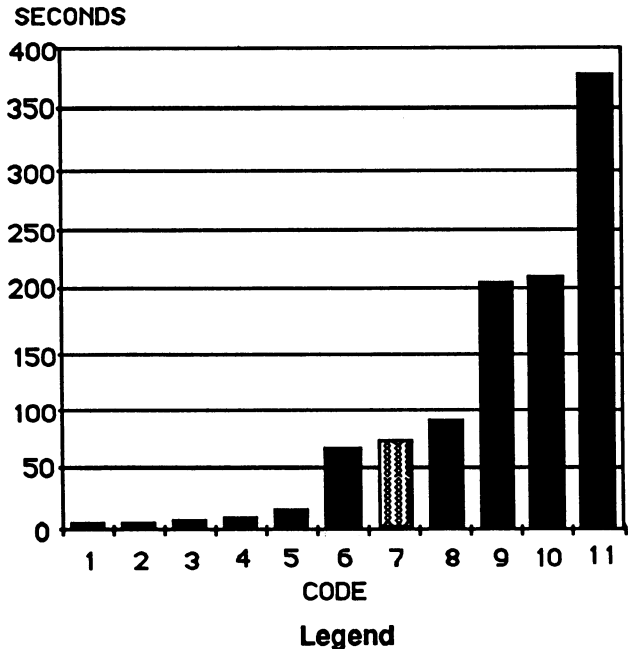
```

 INC(k,prime)
 END;
 INC(count)
 END;
END;
END;
END;
WriteLn; WriteCard(count,6); WriteString(" Primes")
END Prime.

```

### Modula-2 Benchmarks

Comparison between versions and machines



| Code         | System     | Seconds |
|--------------|------------|---------|
| 1. Lilith    |            | 4.24    |
| 2. Hamburg   | Vax 11/750 | 4.64    |
| 3. Logitech  | Eagle      | 6.90    |
| 4. ETH       | LSI-11     | 8.78    |
| 5. Logitech  | IBM PC     | 15.80   |
| 6. Volition  | Sage       | 66.14   |
| 7. Modula    | Macintosh  | 71.00   |
| 8. MRI       | Eagle      | 90.70   |
| 9. MRI       | IBM PC     | 203.60  |
| 10. Volition | IBM PC     | 209.50  |
| 11. Volition | Apple II   | 377.00  |

Source: "Seven Modula-2 Compilers Reviewed" by Terry L. Anderson, *Journal of Pascal, Ada & Modula-2*, Vol. 3 No. 2





# Engineer's Look at Modula-2

## Matrix Multiplication in Modula-2



Jörg Langowski  
MacTutor Editorial Board  
MacTutor Vol. 1 No. 6

The 'almost released' beta version of the Modula-2 compiler by Modula Corporation has been around for some time now on several peoples' desks, including mine. I feel it is time to report some of my experiences with this product.

Modula 2 is in some way the successor of Pascal, created by the same author, Niklaus Wirth from Zürich. Simple Modula 2 programs almost look like Pascal programs. Supposedly some of the inconveniences of Pascal have been straightened out in Modula 2, for example, the rigid order in which declarations are made in Pascal has been relaxed. The most important new concept is that of modular compilation, which means that external (library) definitions can be 'imported' from separately compiled 'modules' at compile time and checked for correct passing of parameters, type mismatches, etc.

But I do not want to talk about language differences here. There has been a whole issue of BYTE dealing with Modula recently, and if you plan to use the Modula-2 compiler, you will have to get the standard reference book, 'Programming in Modula-2', by Niklaus Wirth, anyway.

The Modula-2 version that I received from Modula Corporation came on one Macintosh disk with a three-ring binder manual. The disk contains:

- 1) The Modula-2 compiler and linker. Both are compiled Modula-2 programs, which means (for this system) that they are not native 68000 code, but rather M-code for a virtual machine, which has to be interpreted.
- 2) The interpreter. The disk contains two versions, one of which is used to run the compiler and linker, the other one to run your compiled Modula-2 program
- 3) System libraries with precompiled modules for I/O, file handling, number conversion, string manipulation and - most important of all - toolbox access.
- 4) A Decode utility to disassemble M-code. I found this of little use since I had no definition of the M-code instructions available.
- 5) Source code of a Graphics Demo and the Edit sample application that is also part of the IM manual.
- 6) Edit, the same editor as in the Asm/Edit system.
- 7) Rmaker, the (in)famous resource maker.

The manual gives a rather concise introduction on how to use the compiler and linker if you want to write a simple program or create a separately compiled module. Beside these instructions, it contains a listing of all the predefined modules that come with the system.

If you want to write a program that does just simple I/O and do not care about the user interface, you can use the standard GrafPort that the system comes up with when you

start your application. Unfortunately, this is the full screen, not a nicely set up window like some other compilers have. You merely get a blank screen terminal emulation; if you want to use a window, you will have to define it through toolbox calls. You might want to take a look at the sample program in Listing 1 now.

This is a short matrix multiplication routine that initializes two matrices and determines the product matrix, then types out one element of the product as a check. You see that the actual calculation comprises about 25% of the total program, the rest is definitions and setting up of the window. Having read some of the other examples in MacTutor, you might not be surprised at that kind of overhead. The main parts of the program are:

1. The IMPORT declarations, which get routines out of precompiled modules; note that not even simple I/O is defined by default, you have to import procedures for any kind of I/O that you want to do.

2. Some TYPE and VAR declarations - similar to Pascal. Integers (signed) and Cardinals (unsigned) are 16 bit by default. If you need a 32 bit integer, you have to use the LongCard TYPE, which is a record with the fields h and l, both CARDINAL. You have to assign them separately.

3. Toolbox PROCEDURE declarations. Unfortunately, there is no way to access a toolbox routine through its trap number directly; the routines are called (presumably) through numbers in another module, and the definitions that you have to use are given in the manual. Most of the toolbox routines (not all of them) are included. The definitions make as much sense to me as they make to you, but they seem to work.

4. The actual program. You will notice there that the string format has to be converted from Modula (zero byte at the end) to Macintosh (character count in front) format.

The program is edited using the well-known Edit program, then passed as a file with the extension .MOD to the compiler. Compilation of this sample program takes 2 min 58 secs, add to that 25 secs to bring up the compiler. The reason for the system being so slow is of course that the compiler is written in M-code, too, and is interpreted. You quickly learn to examine your code *very* carefully for errors before you call up the compiler! One rather annoying feature of Modula-2 did not make things easier, either: upper and lower case are distinguished, and it makes a difference whether you write WindowPtr or windowptr. Debugging the example, after it 'almost' ran, took one hour, most of that time waiting for the compiler to finish.

The compiled program is then linked with the library modules (no linker errors here, since external references are already checked for completeness by the compiler) and you get a .LOD file, which, when double clicked, is executed by the Modula runtime interpreter.

Even though the compiler is slow, the generated code is rather fast. The example takes 21 seconds to execute. This is faster as you can ever do with the built-in 80-bit floating point package (judged from a similar program in Forth, which took over 40 seconds, most of that being floating point time). Modula-2 has its own 32-bit floating point package built in,

which is precise enough for most applications. The Sieve runs 9 seconds per iteration, making a total of 90 secs for the standard 10 iterations. This, in turn, is slower than Forth (probably because the ratio M-code to built-in 68000 routines is higher in this case).

The timing results, in my opinion, make the Modula-2 system a very satisfactory choice for scientific and engineering applications. If Modula Corporation could come up with a compiler written in machine code, program development would be much more fun, too.

#### **MODULE Matmul;**

```
FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;
FROM Terminal IMPORT ClearScreen;
FROM ReallnOut IMPORT ReadReal, GWriteReal;
FROM QuickDrawTypes IMPORT Rect, GrafPtr, Str255, LongCard;
FROM ToolBoxTypes IMPORT WindowPtr, WindowRecord;
FROM Strings IMPORT StrModToMac;
FROM SYSTEM IMPORT ADDRESS;
```

```
CONST CX = 355B; QuickDrawModNum = 1; ToolBoxModNum = 2;
```

```
TYPE Ptr = ADDRESS;
```

```
VAR i,j,k : CARDINAL;
 sum,p,q : REAL;
 a : ARRAY [1..20] OF ARRAY [1..30] OF REAL;
 b : ARRAY [1..30] OF ARRAY [1..20] OF REAL;
 c : ARRAY [1..30] OF ARRAY [1..30] OF REAL;
 Wbounds : Rect;
 Wtitle : Str255;
 Wnew : WindowPtr;
 WrefCon, behind, OnHeap: LongCard;
```

```
PROCEDURE SetPort (port : WindowPtr;
 CODE CX; QuickDrawModNum; 4 END SetPort;
```

```
PROCEDURE PortSize (width, height : INTEGER);
 CODE CX; QuickDrawModNum; 8 END PortSize;
```

```
PROCEDURE TextFont (font : INTEGER);
 CODE CX; QuickDrawModNum; 32 END TextFont;
```

```
PROCEDURE TextSize (size : INTEGER);
 CODE CX; QuickDrawModNum; 35 END TextSize;
```

```
PROCEDURE NewWindow (wStorage : Ptr; boundsRect : Rect;
 title : Str255; visible : BOOLEAN;

 theProc:INTEGER; behind : WindowPtr;

 goAwayFlag:BOOLEAN; refCon:LongCard) : WindowPtr;
 CODE CX; ToolBoxModNum; 50 END NewWindow;
```

```
PROCEDURE DrawGrowIcon (theWindow : WindowPtr);
 CODE CX; ToolBoxModNum; 64 END DrawGrowIcon;
```

```
PROCEDURE SelectWindow (theWindow : WindowPtr);
```

CODE CX; ToolBoxModNum; 56 END SelectWindow;

**BEGIN**

**WITH WrefCon DO h := 0; l := 0 END;**

**WITH behind DO h := 65535; l := 65535 END;**

**WITH OnHeap DO h := 0; l := 0 END;**

**WITH Wbounds DO top := 50; left := 20; bottom := 300; right := 500 END;**

**StrModToMac (Wtitle, "Demo Window"); TextFont(0); TextSize(12);**

**Wnew := NewWindow(Ptr(OnHeap),Wbounds,Wtitle,TRUE,0,**

**WindowPtr(behind),TRUE,WrefCon);**

**SetPort(Wnew); SelectWindow(Wnew); DrawGrowIcon(Wnew);**

**PortSize(464,234); ClearScreen; TextFont(3); TextSize(9);**

**WriteString("Initialization values for a: "); ReadReal(p);**

**WriteString(", and b: "); ReadReal(q); WriteLn;**

**FOR i:= 1 TO 20 DO**

**FOR j:= 1 TO 30 DO**

**a[i][j] := p; b[j][i] := q**

**END**

**END;**

**WriteString ("Initialization done."); WriteLn;**

**FOR i:= 1 TO 30 DO**

**FOR j:= 1 TO 30 DO**

**sum := 0.0;**

**FOR k := 1 TO 20 DO sum:=sum + a[k][i]\*b[j][k] END;**

**c[i][j] := sum**

**END**

**END;**

**WriteString ("Multiplication completed."); WriteLn;**

**WriteString (" C[15,16] = "); GWriteReal (c[15][16],10);**

**ReadReal(p)**

**END Matmul.**



# *Modula-2 Mods*

## *Towers of Hanoi, Part 1*



*John Bogan*  
*MacTutor Contributing Author*  
*MacTutor Vol. 1 No. 6*

Often I think of computing as an extraordinary, real-life adventure game beckoning the computer warrior to descend into a yawning cavern riddled with many dark and intriguing passageways, each begging to be explored. At the end of each corridor there lies a treasure, usually small and mostly ordinary, but sometimes great and wonderful. These treasures are jewels of knowledge and enlightenment about the marvels of logic.

I honestly believe that the computing community has barely scratched the surface of our particular adventure.

It is up to each of us to decide which passageways we will pursue. The corridors I have explored for over half a decade are low-cost, professional microsystems, the Motorola 68000, and Pascal. Clues I had accumulated over the years told me there would be something special waiting when these paths crossed. It now occupies the position of honor on my desk, and is the Apple Macintosh.

The reason the Mac is so valuable is that it sheds piercing lights such as "user friendly", "software integration" and "computing for the masses" on gloomy corridors. As an early Macintosh enthusiast who has eagerly watched the hi-res graphics, mice, windows, icons, pulldown menus and cut and paste integration reach acceptance and imitation over the past year, I could almost feel that the Mac is the product of wizards and elfin folk who possess great magic. (One look at Burrell Smith proves that).

### **Deep Background**

It seems to be one of the best kept secrets in Computer Science and Software Engineering (CSSE) that the proper role of High Level Languages (HLLs) is to permit and encourage wide-ranging experimentation and evaluation of new ideas in a cost-efficient and timely fashion. Since learning the difference between FORTRAN and COBOL, I've followed and participated in the religious wars over which computer language is Best, and have concluded that most of the arguments are largely irrelevant, since they concentrate on the question: What language should our software be written in when it is shipped to the consumer?

As someone who is interested in exceptional software requiring extensive R&D, I have consigned that question to the trash can. People who focus on maximizing speed and space insist on Assembler (ASM). Puzzle fanatics choose Forth, and compromisers who can't give up the chance to get their hands dirty with a little register optimization will probably choose "C". But scientists and engineers in R&D labs who attempt to conserve the scarce R&D dollar and even scarcer R&D minute often choose Modula-2.

### **Here's Why**

High level languages are invaluable because they permit software engineers to build complex, innovative programs which break new ground in the shortest possible time, for the fewest R&D dollars. This means within a given budget and deadline, more ideas can be explored and evaluated and thus greater progress made toward achieving an ideal piece of software. Let's look at some of the most important software breakthroughs in the past few years and examine the language influences on the R&D of these milestone products:

**CP/M.** Designed largely in PL/M, a structured HLL. Responsible for the early success of 8080 and Z-80 S-100 small business systems.

**Visicalc.** Designed largely in BASIC, and responsible for the immense success of the Apple //.

**Lotus 1 2 3.** Cloned from MBA Context, which was written in UCSD Pascal. Responsible for the flood of integrated software and the dominance of the IBM PC.

**UCSD Pascal P-system.** Contributed to pull down menus and menu driven Operating Environments.

**Xerox Alto testbed.** This system never made it into the commercial world at all, yet has to be considered the grandfather of the graphic, iconic, windowed personal workstation. This system was developed in the HLL Mesa, a Pascal derivative.

**Apple LISA.** The first commercial microcomputer to make it to the marketplace bearing the fruit of the Xerox Alto, LISA was designed and written in Pascal.

**Apple Macintosh.** Designed in Pascal and translated into hand-optimized ASM. The third milestone microcomputer, the most successful introduction of any professional micro at any time (in my opinion).

Kildal's GEM and Tramiel's "Jackintosh", otherwise known as Macintosh clones.

These are historical facts which cannot be ignored. They illustrate the major role of HLLs, in particular the structured language Pascal, in software R&D labs throughout the micro world. The reason for success is obvious and worth restating: with a structured HLL, it is easy to break new ground and if this new ground is genuinely helpful and makes small computers more productive and easier to master, then consumers will eat them up.

Why are structured languages going to be the light of choice in R&D labs of the micro world for the indefinite future? The answer lies in history, economics and logic.

## Programming as Engineering

Back in the mid-1960's, computer specialists noticed a troubling trend. Although hardware was becoming exponentially cheaper, the cost of producing software was rising, as programs became more complex to take advantage of the improved hardware. Figure 1 illustrates this point.

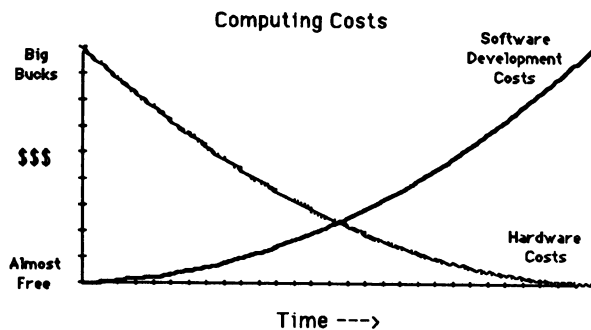


Figure 1

Following the realization that software costs were getting out of hand, the computer industry began diverting resources from the hardware side of the industry to the software side. Engineering entered the picture. A rough definition of engineering is the art of measuring, and of optimizing resources.

And when the productivity boys came into data processing departments with their legal pads and stopwatches, the unit of productivity measure they decided on was the number of lines of code per day per programmer. To their horror, here is what they found:

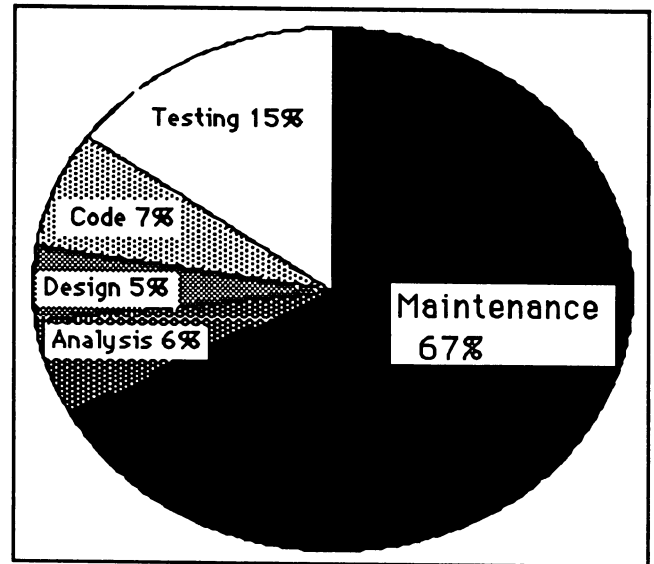
1) The average programmer cranked out only ten lines of debugged code a day, and 2), it didn't matter what language the programmers were coding in! If they were programming in Assembly, the daily output was ten ASM statements. If the language was COBOL or FORTRAN - still ten statements!

The first, and obvious, conclusion drawn was that COBOL and FORTRAN programmers were 10 to 100 times more productive than ASM programmers, since each line of FORTRAN could generate ten lines of ASM, and some lines of COBOL could generate 100 ASM statements. (I believe that these widely reported results are in no small measure responsible for the fact that most large DP departments are now and forever locked into COBOL as their language of choice. So much for the benefits of software engineering!)

But before dismissing software engineering, let's see what else was learned in the productivity studies.

When you ask a talented hacker how long it took to write a 1000-line program, don't be surprised if he scratches his beard and replies, "Oh, I did that one night a couple of months ago." This seems to contradict the 10-statement-per-day finding by orders of magnitude. But then ask the hacker how long he spent getting his program bug-free. He'll probably say "Well, I expect to get the last bug out of it anytime now."

One night writing the code and two months debugging it is more common than you might think. Figure 2 illustrates the traditional division of labor (hence cost) in creating significant programs.



A careful look at Figure 2 reveals how *obscene* it is. In traditional programming, two thirds of the effort and money goes down the toilet known as maintenance. Maintenance means three things:

1. Fixing bugs
2. Making late changes because of poor initial analysis and design
3. Making changes due to unanticipated user requirements

Figure 2 indicates why the 10 lines-per-day rule is largely true. The actual coding of a program is almost insignificant in comparison to revisions of the code that begin once the code is "completed". This frightening discovery was made in the mid-60's and to this date remains the driving force behind most of the effort going into software engineering.

### External Modularity

In travels from computer center to computer center, one of the early discoveries made by software engineers was that thousands of lines of code were being duplicated by each of the various DP staffs. An obvious solution was proposed: libraries of code should be maintained and shared among computer programmers. This concept was termed **external modularity**, and after some half-hearted attempts to implement this seemingly worthwhile idea in COBOL, the sharing of code promptly washed ashore and soon began to smell like the dead fish it was. The failure of modularity on the library scale was due to the fact that programmers functional illiterates when it comes to documenting their code. Since reading someone else's code is frequently more difficult than writing the same program from scratch, most attempts at

libraries never made it past the circular file. Also, COBOL is a pathetic language for attempting to support libraries. At last a common syndrome known as NIH: Not Invented Here was, and still is, common among computer folk. NIH is a question of pride... "I can do it better than he can do it, so we will scrap his version and start over from the beginning". Modularity at the library level was consigned to the dust heap around 1970.

### What Is This Garbage?

(Or: I thought this was going to be a column about Modula-2 and the Mac!)

This column began with an analogy to adventure. I would like to end it by returning to the Cave of Computing. This journal - *MacTutor* - is about teaching all comers how to program the Macintosh in a standalone environment.

Did you know that programming the Mac is at least an order of magnitude more difficult than programming any other small computer? That is because the Mac ROM and the user interface are so rich (read, complex). One look at the Mac programming manual Inside Macintosh should convince you of this. This complexity is the main reason why Mac software has been as slow in appearing as it has. Everyone is rethinking what it means to write a quality program with a friendly 'face'. In cave adventure terms, everyone is stumbling around the twisty little maze knowing that enormous treasure lies within spitting distance. But to get out of the maze you need...the magic tools! Modula-2 and software engineering.

Talking about programming the Mac in Modula-2 without talking about modularity is absurd. Talking about modularity without talking about software engineering is absurd. Talking about software engineering without talking about Modula-2 is absurd. This context is the twisty little maze I propose to map in the rest of this article.

### For Example

Here is a concrete example of why the interaction between these topics is so important. Earlier in this column, the problem of sharing code between installations and programmers was referred to as **external modularity**. Around 1970 this idea died, leaving behind a big stink. But in January 1984, Steve Jobs and friends resurrected it from the grave. *What else is the Mac Toolbox ROM but a massive external library with 480 entry points?* With this library, ordinary programmers are encouraged to share the fully optimized code of some of the world's best programmers. This library is a construction set for the finest user interface the industry has yet witnessed.

What fewer people know is that Niklaus Wirth anticipated Steve Jobs and built a tool, Modula-2, that would be capable of harnessing the power of those two ROMs. When I spoke on the phone to David Smith about submitting a column on Modula-2 and the Mac, I mentioned in passing that I brought a lot of baggage to the subject. This baggage is

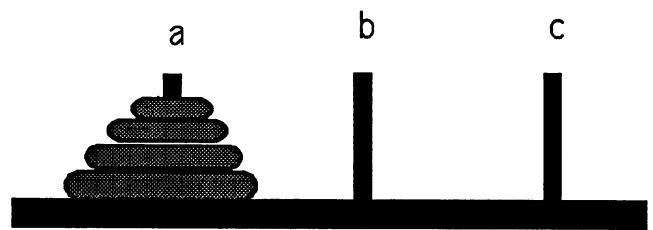
software engineering. If you think this is appropriate fare for *MacTutor*, write to me via the journal and share your thoughts.

## A Programming Testbed

The following program, The Towers of Hanoi, will be our programming project for the next  $n$  columns. It is a classic computer science example usually reserved for discussions of non-tail recursion. The purpose will be to play with Quickdraw and the Mac interface. Since my graphic abilities peaked at bad stick figures in the third grade, anyone who submits a better version than mine will get his or hers printed in future columns. The version below is text only and doesn't yet use the Mac library, but I expect it to grow into a worthwhile graphics program within a few short columns.

Towers of Hanoi is simple. You start with three vertical pegs: call them a, b and c. In the starting position, peg a contains a stack of disks, the largest on the bottom and the smallest on the top. (The number of disks may range from three to nine, but our simple example will use four.) Object of the game is to move the disks, one at a time, among the pegs until the original stack is recreated on peg c. The one rule is, at no time may a larger disk rest on top of a smaller. That's it. Figure 3 may make this a little more clear.

The Towers of Hanoi



MODULE HanoiPuzzle;

```
(* Declare I/O from Modula-2 Standard Library *)
FROM Terminal IMPORT ClearScreen;
FROM InOut IMPORT WriteLn, WriteString, WriteCard,
ReadCard,
Write;
```

```
CONST Start = "a";
 Int = "b";
 Finish = "c";
```

```
VAR DiskCount: CARDINAL; Done: BOOLEAN;
```

```
(* Get number of disks or set terminate flag *)
PROCEDURE GetInput (VAR NumberOfDisks: CARDINAL;
VAR Quit: BOOLEAN);
```

```
BEGIN
```

```
 ClearScreen;
 WriteString("Enter number of disks (between 3 and 9)");
 WriteLn;
 WriteString("To quit - enter number out of range");
 WriteLn;
 ReadCard(NumberOfDisks);
```

```

IF (NumberOfDisks < 3) OR (NumberOfDisks > 9)
 THEN Quit := TRUE
 ELSE Quit := FALSE
END; (*IF*)
ClearScreen;
END GetInput;

```

```

(* The recursive guts of the programs ... calculate moves. *)
PROCEDURE Hanoi(n: CARDINAL; StartNeedle, IntNeedle,
 FinishNeedle: CHAR);

```

```

BEGIN
 IF n#0
 THEN
 Hanoi(n-1, StartNeedle, FinishNeedle, IntNeedle);
 WriteLn;
 WriteString("Move disk -");
 WriteCard(n,2);

```

```

 WriteString(" from ");
 Write(StartNeedle);
 WriteString(" to ");
 Write(FinishNeedle);
 Hanoi(n-1, IntNeedle, StartNeedle, FinishNeedle);
 END; (*IF*)
END Hanoi;

```

```

(* Mainline ... control main loop ... get input & do it. *)
BEGIN
 GetInput(DiskCount, Done);
 WHILE NOT Done DO
 Hanoi(DiskCount, Start, Int, Finish);
 GetInput(DiskCount, Done);
 END; (*WHILE*)
END HanoiPuzzle.

```



# Modula-2 Mods

## Towers of Hanoi, Part 2



John Bogan  
MacTutor Contributing Author  
MacTutor Vol. 1 No. 7

This month we'll explore three things: first, a continued introduction to the elements of software engineering in a historical perspective; secondly, why Modula-2 is not a hackers' language, and finally we'll look at a sample program that uses the Mac ROM to draw the starting position for the Towers of Hanoi.

### A New Direction

Last month we saw that the field of software engineering is capable of abuse, as well as having the ability to give important insights into building good computer software. By the late 1960's, software engineers seemed, in effect, to have dictated that COBOL would be the primary language of large companies for a long time to come. Microcomputer programmers faced with the prospect of coding in COBOL shuddered in terror at the thought.

Then in 1968, software engineering took a turn for the better when a famous, well-regarded European computer scientist by the name of E.W. Dijkstra lit a fire under the COBOL and FORTRAN programming community. Dijkstra wrote a letter to the *Communications of the ACM* entitled "GOTO Statement Considered Harmful". Dijkstra observed, after observing a multitude of programs in a variety of languages, that the quality of a program was inversely proportional to the number of GOTO statements in that program.

It's a simple idea. Jumping around a program with branching statements leads to unreadable program texts (known in the trade as spaghetti code), which are next to impossible to debug, or for a third party to pick up and read. Since ALGOL-60, Europe's favorite language, has advanced control structures permitting GOTO-less programming, and FORTRAN doesn't, battle lines were drawn and letter after letter poured into computer journals, feeding the flames.

In an effort to quell the controversy, Dijkstra, together with Dahl and Hoare, finally published a book on how to write high-quality programs without using GOTO statements. This book *Structured Programming*, published in 1972 established once and for all that GOTOs are redundant. Every sequential programming task can be accomplished with a combination of three constructs:

- **Sequence:**  
BEGIN ... s1 ... s2 ... s3 ... END
- **Iteration:**  
WHILE c1 DO ... s1 ... ENDWHILE
- **Conditional:**  
IF c1 THEN s1 ENDIF

The principles of structured programming enjoyed great success, and the course of software engineering was changed forever. One example of this change is the language Modula-

2, which is rich in structured control statements and does not support the GOTO at all. There are no statement labels in Modula-2, and while it is possible to write poor code in Modula, it is impossible to write spaghetti code.

Structured control statements supported by Modula are the statement sequence, the WHILE ... DO, the REPEAT ... UNTIL, the FOR ... TO ... BY ... DO, the LOOP ... EXIT, the IF ... THEN ... ELSIF, the CASE ... OF and the WITH ... DO statements.

### What Is Structured?

In some ways this is difficult to answer. For adherents of structured techniques, it might be defined as the following: a philosophy for solving problems which attempts to conserve scarce resources by arriving at the perfect solution in the fewest attempts by following a plan.

### Why Plan When You Can Hack?

The idea of a plan is very important in understanding structured techniques, software engineering and Modula-2. Most hackers I've known use the "technique of incremental discovery", or trial by error. Programs just grow from line 1 until the last bell and whistle is debugged. Assembly language, and to a lesser extent C, are well-suited for this type of programming. Modula-2 most definitely is not. The quality of a Modula-2 program is dependent on the quality of detailed planning that occurs before the first line of code is written. In many ways this dependence on upfront planning is a distinct disadvantage for learning a new and unique system like the Mac. So many of the techniques peculiar to the Mac, such as the entire user interface, resources or Quickdraw, are best approached and mastered by trial-and-error hacks. When you consider the compile-link-execute overhead of Modula-2, a good strategy for making optimal use of it appears to be learning the Mac with Apple's interpreted Pascal, and then translate these programs into the much faster Modula-2. Modula and Pascal are actually quite close to one another.

### This Month's Code

The Modula-2 code that follows shows how to access the Mac ROM on a 128K machine. Quickdraw calls used are SetRect, PaintRect and PaintRoundRect. (Be aware that the method of specifying ROM calls is different for a 512K). Also note that the data types VHSelect, Point, and Rect could have been imported blindly instead of spelled out, but then their internal structures would have been hidden.



```

MODULE Hanoi;
(* build starting position for Hanoi Towers *)

FROM Terminal IMPORT ClearScreen;
FROM InOut IMPORT WriteString, ReadCard, WriteLn;

(* data structures for Quickdraw calls *)
TYPE
 VHSelect = (v,h);

 Point = RECORD
 CASE INTEGER OF
 0: v: INTEGER;
 h: INTEGER;

 |1: vh: ARRAY VHSelect OF INTEGER;
 END; (* CASE *)
 END; (* RECORD *)

 Rect = RECORD
 CASE INTEGER OF
 0: top: INTEGER;
 left: INTEGER;
 bottom: INTEGER;
 right: INTEGER;

 |1: topLeft: Point;
 botRight: Point;
 END; (* CASE *)
 END; (* RECORD *)

CONST
 CX = 355B;
 QuickDraw1ModNum = 2; (* absolute module number
 of QuickDraw1 *)

VAR
 r: Rect; NumDisks: CARDINAL;

PROCEDURE SetRect (VAR r: Rect; left,top,right,bottom:
INTEGER);
 CODE CX; QuickDraw1ModNum; 51 END SetRect;

PROCEDURE PaintRect (r: Rect);
 CODE CX; QuickDraw1ModNum; 62 END PaintRect;

PROCEDURE PaintRoundRect(r: Rect; ovWd, ovHt:
INTEGER);
 CODE CX; QuickDraw1ModNum; 67 END PaintRoundRect;

PROCEDURE DrawBase;
 CONST
 BaseLeft = 36;
 BaseTop = 261;
 BaseRight = 476;
 BaseBottom = 270;

 BEGIN
 SetRect(r,BaseLeft,BaseTop,BaseRight,BaseBottom);
 PaintRect(r);
 END DrawBase;

PROCEDURE DrawPosts;
 CONST

```

```

 PostTop = 144;
 PostBottom = 261;
 PostWidth = 6;
 HalfPostWidth = PostWidth DIV 2;
 PostPosition = 128;

 VAR
 n, PostLeft, PostRight: INTEGER;
 BEGIN
 n:=1;
 WHILE n <= 3 DO
 PostLeft := (PostPosition * n) - HalfPostWidth;
 PostRight := PostLeft + PostWidth;
 SetRect(r,PostLeft,PostTop,PostRight,PostBottom
);
 PaintRect(r);
 n:= n + 1;
 END; (* WHILE *)
 END DrawPosts;

PROCEDURE DrawVarDisks(numberofdisks: CARDINAL);
 CONST
 bigdiskleft = 128 - 60;
 bigdisktop = 261 - 12;
 bigdiskright = 128 + 60;
 bigdiskbottom = 261;
 deltalength = 5;
 deltadepth = 12;
 VAR leftedge, topedge, rightedge, bottomedge: INTEGER;
 i: CARDINAL;
 BEGIN
 IF (numberofdisks > 2) AND (numberofdisks < 10)
 THEN
 leftedge := bigdiskleft; topedge := bigdisktop;
 rightedge := bigdiskright; bottomedge :=
bigdiskbottom;
 SetRect(r,leftedge,topedge,rightedge,bottomedge
);
 PaintRoundRect(r,40,40);
 FOR i := 1 TO numberofdisks - 1 DO
 leftedge := leftedge + deltalength;
 topedge := topedge - deltadepth;
 rightedge := rightedge - deltalength;
 bottomedge := bottomedge - deltadepth;
 SetRect(r,leftedge,topedge,rightedge,bottomed
ge);
 PaintRoundRect(r,40,40);
 END; (* FOR *)
 END; (* IF *)
 END DrawVarDisks;

PROCEDURE GetInput(VAR NDisks: CARDINAL);
 BEGIN
 ClearScreen;
 WriteString("Enter number of disks (between 3 to 9)");
 WriteLn;
 WriteString("To quit - enter number out of range");
 ReadCard(NDisks);
 ClearScreen;
 END GetInput;

PROCEDURE InitGraphics(NumberOfDisks: CARDINAL);
 BEGIN

```

```
DrawBase;
DrawPosts;
DrawVarDisks(NumberOfDisks);
END InitGraphics;

PROCEDURE ExecuteTowers;
VAR Delay: CARDINAL;
BEGIN
 FOR Delay := 1 TO 30000 DO END; (* FOR *)
END ExecuteTowers;
```

```
BEGIN
 GetInput(NumDisks);
 WHILE (NumDisks >= 3) AND (NumDisks <= 9) DO
 InitGraphics(NumDisks);
 ExecuteTowers;
 GetInput(NumDisks);
 END; (* WHILE *)
END Hanoi.
```



# Programmer's Forum

## Modula-2 and Anchored Variables



Tom Taylor  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 8

The programming language Modula-2 supports a little known but extremely powerful feature called "anchored variables." Anchored variables allow one to specify in the variable's declaration the absolute address of the variable and override the compiler's allocation method. This feature was intended to allow access to device registers on computers with memory mapped I/O (like on a PDP-11 where the original Modula-2 compiler was developed). It also allows the Modula-2 programmer on the Macintosh to access the Mac's operating system global variables very conveniently and cleanly.

In this article, we will look at two examples of anchored variable usage. In the first, we will try a small example to show that anchored variables really do work. In the second example, we'll actually develop a useful module based on anchored variables. These examples have been programmed using MacModula-2 by Modula Corporation. However with minor changes to the programs (probably the import statements), these programs should run using any Modula-2 system.

Anchored variables are "anchored" to a specific address by specifying that address in brackets immediately following the variable name in a variable declaration and before specifying the variable's type. For example:

```
VAR
 FinderName [2E0h]
 : ARRAY [0..16] OF CHAR;
```

This statement tells the compiler to allocate the variable *FinderName* starting at hex location 2E0h. Location 2E0h just happens to contain a Mac-style string with the name of the current finder. The following program will print out the name of the currently installed finder:

```
MODULE PrintFinder;
 FROM Terminal IMPORT
 ClearScreen, Write,
 Read, WriteLn, WriteString;

 VAR
 FinderName [2E0h] :
 ARRAY [0..16] OF CHAR;
 i : CARDINAL;
 ch : CHAR;

 BEGIN
 ClearScreen;
 WriteString("Current finder is: ");
 FOR i := 1 TO ORD(FinderName[0]) DO
 Write(FinderName[i]);
```

```
END;
WriteLn;
Read(ch);
END PrintFinder.
```

### Volume Control Block Queue

This next example is slightly more useful. It demonstrates the use of anchored variables in traversing the Volume Control Block queue and returning information about any disk (or volume) visible on the desktop (inserted in a drive or ejected) at the time your program was launched. This technique is used, for example, in the MacModula-2 compiler and linker when they search for imported files. This feature allows users of single drive Macs to build programs that are spread out over a number of disks by having those disks visible on the desktop at the time the compiler or linker is launched. The compiler or linker will search each disk found on the desktop by traversing the VCB queue until the desired import file is found or the end of the queue is reached.

The following example actually consists of three modules:

- 1) A **Definition Module** that defines the records, types, global variables, and procedures that are available for use by any program.
- 2) An **Implementation Module** that contains the code for the procedures defined in the Definition Module.
- 3) A simple **Program Module** that uses the procedures we have written and demonstrates some of the information available from the Volume Control Blocks.

```
DEFINITION MODULE VolumeTracer;

FROM MacSystemTypes IMPORT LongCard, Ptr;

EXPORT QUALIFIED VCB, VolumesOnLine,
GetVolumeInfo;

TYPE
 QElemPtr = POINTER TO VCB;

VCB = RECORD
 qLink: QElemPtr;
 (* next queue entry *)
 qType: INTEGER;
 (* not used *)
 vcbFlags: INTEGER;
 (* bit 15=1 if dirty *)
 vcbSigWord: INTEGER;
```

```

 (* always $D2D7 *)
vcbCrDate: LongCard;
 (* date volume initialized *)
vcbLsBkUp: LongCard;
 (* date of last backup *)
vcbAtrb: INTEGER;
 (* volume attributes *)
vcbNmFis: INTEGER;
 (* # of files in directory *)
vcbDirSt: INTEGER;
 (* directory's first block *)
vcbBILn: INTEGER;
 (* length of file directory *)
vcbNmBlks: INTEGER;
 (* # of allocation blocks *)
vcbAIBlkSiz: LongCard;
 (* size of allocation blocks *)
vcbClpSiz: LongCard;
 (* # of bytes to allocate *)
vcbAIBSt: INTEGER;
 (* first block in block map *)
vcbNxtFNum: LongCard;
 (* next unused file number *)
vcbFreeBks: INTEGER;
 (* number of unused blocks *)
vcbVN: ARRAY [0..27] OF CHAR;
 (* vol name Str255 format *)
vcbDrvNum: INTEGER;
 (* drive number *)
vcbDRefNum: INTEGER;
 (* driver reference number *)
vcbFSID: INTEGER;
 (* file system identifier *)
vcbVRefNum: INTEGER;
 (* volume reference number *)
vcbMAdr: Ptr;
 (* location of block map *)
vcbBufAdr: Ptr;
 (* location of volume buffer *)
vcbMLen: INTEGER;
 (* # of bytes in block map *)
vcbDirIndex: INTEGER;
 (* used internally *)
vcbDirBlk: INTEGER;
 (* used internally *)

```

END;

```

PROCEDURE VolumesOnLine(): CARDINAL;
 (* Returns the maximum number of
 volumes currently recognized by the
 Mac operating system. *)

```

```

PROCEDURE GetVolumeInfo(VAR volume :
 VCB; whichVol : CARDINAL);
 (* Returns the current VCB block for
 volume "whichVol" The variable
 "whichVol" must be between 1 and
 the result of the procedure of
 "VolumesOnLine()". Otherwise,
 the "volume" info is undefined. *)

```

END VolumeTracer.

IMPLEMENTATION MODULE VolumeTracer;

TYPE

```

 QHdrPtr = POINTER TO QHdr;
 QHdr = RECORD
 qFlags : INTEGER;
 (* queue flags *)
 qHead : QElemPtr;
 (* first queue entry *)
 qTail : QElemPtr;
 (* last queue entry *)

```

END;

VAR

```

 VCBQHdr [0356h] : QHdr;
 (* VCB queue header *)

```

PROCEDURE VolumesOnLine(): CARDINAL;

VAR

```

 ptr : QElemPtr;
 count : CARDINAL;

```

BEGIN

```

 ptr := VCBQHdr.qHead;
 count := 0;
 WHILE ptr # NIL DO
 INC(count);
 ptr := ptr^.qLink;

```

END;

RETURN count;

END VolumesOnLine;

```

PROCEDURE GetVolumeInfo(VAR volume :
 VCB; whichVol : CARDINAL);

```

VAR

```

 ptr : QElemPtr;
 count : CARDINAL;

```

BEGIN

```

 ptr := VCBQHdr.qHead;
 count := 0;
 WHILE (ptr # NIL) AND
 (count # whichVol) DO
 INC(count);
 IF count = whichVol THEN
 volume := ptr^;
 END;

```

ptr := ptr^.qLink;

END;

END GetVolumeInfo;

END VolumeTracer.

MODULE ListVolumes;

FROM VolumeTracer IMPORT

VCB, VolumesOnLine, GetVolumeInfo;

FROM InOut IMPORT WriteString, ClearScreen,  
WriteLn, WriteCard, WriteInt, Write, Read;

VAR

```

 i, maxVols : CARDINAL;
 vcb : VCB;
 ch : CHAR;

```

```

PROCEDURE PrintVolName;
VAR
 i : CARDINAL;
BEGIN
 FOR i := 1 TO ORD(vcb.vcbVN[0]) DO
 Write(vcb.vcbVN[i]);
 END;
 WriteLn;
 END PrintVolName;

BEGIN
 ClearScreen;
 WriteString
 ("Number of volumes on-line: ");
 maxVols := VolumesOnLine();
 WriteCard(maxVols,0);
 WriteLn; WriteLn;
 FOR i := 1 TO maxVols DO
 GetVolumeInfo(vcb,i);
 WriteString('Volume Name: ');
 PrintVolName;
 WriteString
 ('Number of files in volume: ');
 WriteInt(vcb.vcbNmFis,0);
 WriteLn;
 WriteString('Drive Number: ');
 WriteInt(vcb.vcbDrvNum,0);
 WriteLn;
 WriteLn;
 END;
 Read(ch);
END ListVolumes.

```

This is only one simple example of a typical use of anchored variables. Many times, *Inside Macintosh* will mention some variable that can be accessed from assembly language. The Window Manager, for example, mentions that putting a WindowPtr in the variable GhostWindow, will cause that window to never be the front window. In order to find the addresses of such variables, such as GhostWindow, one need simply paw through ToolEqu.TXT or SysEqu.TXT. Both of

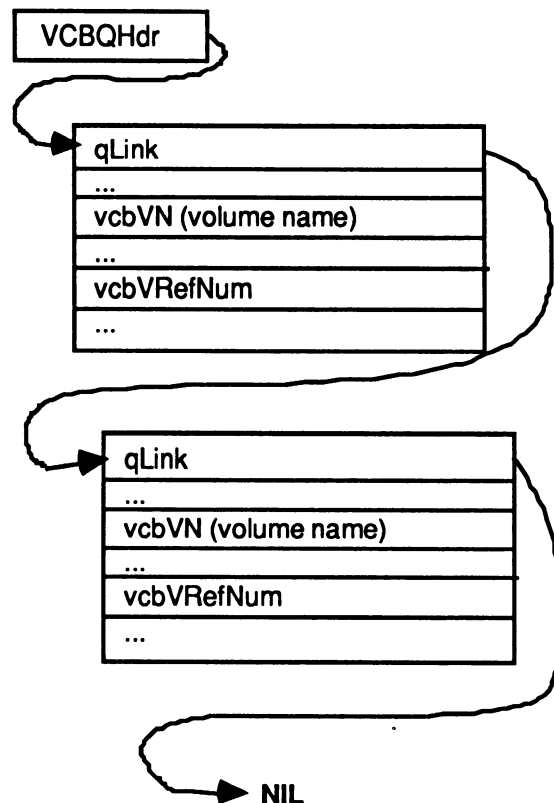


**Figure 1. Typical VCB Queue**

these source files are included with the MDS system by Apple. Not only have I used GhostWindow as an anchored variable in my applications, I have also used anchored variables to access the information set by the Finder when a program is launched.

By taking advantage of the power of anchored variables, you will be able to create very readable programs that use some of the Mac's low-level features.



# Modula 2 Mods

## Using the Alternate Screen Buffer



Tom Taylor  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 9

This article discusses the "alternate screen" capabilities of the Macintosh and presents a module that allows easy access to this facility from MacModula-2.

### The Alternate Screen

As far as I know, *Inside Macintosh* only mentions the alternate screen buffer in two places: in the *Memory Manager Programmer's Guide* and in the *Segment Loader Programmer's Guide*. The Memory Manager section simply says, "There are alternate screen and sound buffers for special applications. If you use either or both of these, the space available for use by your application is reduced accordingly..." The alternate screen buffer can be used for high-speed flicker free graphics or a slide show program. For example, an application can be drawing in one screen while displaying a different screen. The popular "Tumbling Mac" demo by Mainstay uses the alternate screen buffer to create a high-speed flicker-free animation application. According to figure 1, the alternate screen buffer is allocated 32768 bytes below the main screen buffer.

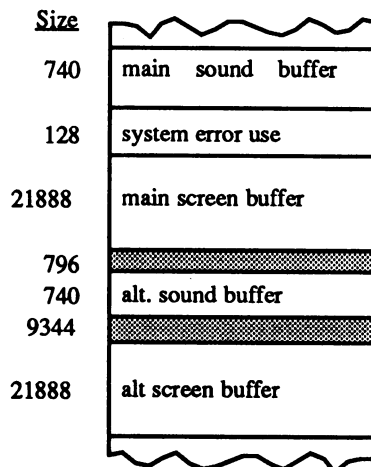


Figure 1: Partial memory map showing sizes of various buffers

The Segment Loader section mentions that the Chain and Launch traps configure memory for the sound and screen buffers. Whenever one of these traps is called, a parameter is passed that determines how these buffers are set up. The word (16-bit) parameter has three possible values:

Zero - Only the main sound and screen buffers are allocated.

This is the normal situation and gives an application the most possible memory.

Negative - The alternate sound buffer and main screen buffer is allocated.

Positive - Both the alternate sound and screen buffers are allocated.

### A Screen Switcher Module

Instead of building a specific program to demonstrate the alternate screen in Modula-2, this article presents a library module that can be used in many different programs in many different types of applications. This article will also present a small program that uses the developed screen switcher module.

A screen switcher module should support at least four functions:

- Launch a program with the alternate screen,
- Copy the main screen into the alternate screen,
- Change the screen back and forth between the main and alternate screens, and
- Set the screen buffer to the main screen before a program exits.

Apparently, the Finder always launches a program with the main sound and screen buffers (parameter set to 0). In order for a program to use the alternate screen, the program must be re-launched with the appropriate parameter (-1). The procedure `LaunchWithAltScreen` re-launches the program with the alternate screen if the alternate screen is not already allocated.

Here are the Definition and Implementation modules for handling the alternate screen:

DEFINITION MODULE ScreenSwitcher;

EXPORT QUALIFIED  
LaunchWithTwoScreens,  
CopyScreens,  
SwitchScreens,  
SetMainScreen;

PROCEDURE LaunchWithTwoScreens;

(\* This procedure re-launches the current program with the alternate screen. If the alternate screen is already installed, this procedure does nothing. \*)

PROCEDURE CopyScreens;

(\* CopyScreens copies the main screen into the alternate screen. \*)

PROCEDURE SwitchScreens;

(\* SwitchScreens alternately switches

what is being displayed from the one screen to the other. \*)

```
PROCEDURE SetMainScreen;
(* Sets the current screen buffer to
the main screen so that when the
program exits to the Finder
the screen will be the right one. *)
```

END ScreenSwitcher.

IMPLEMENTATION MODULE ScreenSwitcher;

```
FROM Launcher IMPORT
 CountAppFiles,
 LaunchRec,
 Launch;
```

```
FROM Terminal IMPORT
 WriteString;
```

```
FROM MacSystemTypes IMPORT
 Str255, LongCard;
```

```
FROM Strings IMPORT
 StrModToMac;
```

```
FROM SYSTEM IMPORT
 ADR, WORD, ADDRESS;
```

```
FROM MacInterface IMPORT
 thePort;
```

```
FROM QuickDraw1 IMPORT
 GrafPtr;
```

```
CONST
 vBufA = 1e00h; (* Buffer A offset from
 VIA interface chip *)
 vPage2 = 1; (* This is a bit into
 vBufA, if 0, select
 alternate screen *)
```

```
TYPE
 ScreenPtr
 = POINTER TO ARRAY [1..10944] OF
 WORD;
 (* The Mac screen buffer is 10944
 16-bit words long *)
```

```
FlagsPtr
 = POINTER TO BITSET;
```

```
VAR
 CurPageOption [936h] : INTEGER;
 (* Mac global variable containing
 value of parameter at launch time
 that determines sound and screen
 buffers. *)
 ScrnBase [824h] : ScreenPtr;
 (* Mac global containing address of
```

```
main screen. *)
 AltBase : ScreenPtr;
 (* Our homebrew alternate screen
 pointer. *)
 VIA6522Chip [1d4h] : FlagsPtr;
 (* VIA base address - from MDS's
 SysEqu.txt *)
 BufferA : FlagsPtr;
 (* Our pointer to vBuffer A *)
 originalPort : GrafPtr;
 (* Save GrafPtr to whole screen *)
```

```
PROCEDURE LaunchWithTwoScreens;
VAR
 PrintOrWhat, count : INTEGER;
 launchRec : LaunchRec;
 prog : Str255;
BEGIN
 IF CurPageOption >= 0 THEN
 CountAppFiles(PrintOrWhat,count);
 (* Get the number of Modula-2
 programs launched. It better
 be one... this one! *)
 IF count # 1 THEN
 WriteString("Double-click the ");
 WriteString('program icon to start');
 WriteString(' the program');
 HALT;
 END;
 StrModToMac(prog,'Modula-2');
 launchRec.Name := ADR(prog);
 launchRec.SoundScreenBuffer := -1;
 Launch(launchRec);
 END;
END LaunchWithTwoScreens;
```

```
PROCEDURE CopyScreens;
BEGIN
 AltBase^ := ScrnBase^;
END CopyScreens;
```

```
PROCEDURE SwitchScreens;
BEGIN
 (* According to the MDS SysEqu.Txt
 file, the vPage2 bit in the BufferA
 flags is zero for the alternate
 screen and one for the main screen.
 *)
 WITH originalPort^.portBits DO
 IF baseAddr = ADDRESS(ScrnBase)
 THEN
 (* Graphical operations apply
 to alternate screen while
 displaying main screen. *)
 baseAddr := ADDRESS(AltBase);
 INCL(BufferA^,vPage2);
 (* Turn on bit *)
 ELSE
 (* Graphical operations apply
 to main screen while displaying
 alternate screen. *)
```

```

 baseAddr := ADDRESS(ScrnBase);
 EXCL(BufferA^,vPage2);
 (* Turn off bit *)
END;
END;
END SwitchScreens;

PROCEDURE SetMainScreen;
BEGIN
 IF NOT(vPage2 IN BufferA^) THEN
 SwitchScreens;
 END;
END SetMainScreen;

BEGIN
 AltBase := ScrnBase;
 (* Set up pointer to alt screen *)
 DEC(AltBase,32768);

 BufferA := VIA6522Chip;
 (* Set up pointer to Buffer A flags *)
 INC(BufferA,vBufA);

 (* Save screen GrafPort *)
 originalPort := thePort;
END ScreenSwitcher.

```

Here is a program module that actually uses the ScreenSwitcher module. The program simply bounces a picture of a Lisa computer around the screen. The program draws the picture in one screen, switches screens, erases the old picture, moves to a new position, draws the picture, and switches to the other screen and applies the same sequence of operations. In order for this program to run properly, it is important to have this program and the Modula-2 program on the system disk. Click the mouse to exit the program.

```

MODULE Animate;

FROM QuickDraw1 IMPORT
 QDPtr, StuffHex, HideCursor,
 BitMap, Rect, SetRect,
 srcOr, srcBic, OffsetRect,
 Random;

FROM QuickDraw2 IMPORT
 CopyBits;

FROM MacSystemTypes IMPORT
 Str255;

FROM Strings IMPORT
 StrModToMac;

FROM SYSTEM IMPORT
 ADR, WORD;

FROM MacInterface IMPORT
 thePort, Write;

FROM EventManager IMPORT

```

Button;

```

FROM ScreenSwitcher IMPORT
 LaunchWithTwoScreens,
 CopyScreens, SwitchScreens;

```

```

VAR
 icon : ARRAY [0..95] OF WORD;
 (* Storage to hold icon data *)
 x, y : ARRAY [0..1] OF INTEGER;
 (* coordinates of icon for each
 screen *)
 screen : [0..1];
 (* current screen reminder *)
 xDir, yDir : INTEGER;
 (* offset for each icon move *)

```

```

PROCEDURE InitIcon;

```

```

 PROCEDURE MyStuffHex(ptr: QDPtr;
 s : ARRAY OF CHAR);
 (* This procedure isolates the body of
 InitIcon from having to convert
 the Modula-style hex strings to
 Pascal format *)

```

```

 VAR
 macStr : Str255;
 BEGIN
 StrModToMac(macStr,s);
 StuffHex(ptr,macStr);
 END MyStuffHex;

```

```

 BEGIN
 (* Picture of a Lisa hoisted from
 Apple's QDSample program *)
 MyStuffHex(ADR(icon[0]),
 '0000000000000000000000000000000001FFFFFFFFFC');
 MyStuffHex(ADR(icon[12]),
 '006000000006018000000000B0600000000130FFFFFFFFFA3')
 ;
 MyStuffHex(ADR(icon[24]),
 '18000000004311FFFFF00023120000080F231200000BF923');
 MyStuffHex(ADR(icon[36]),
 '120000080F23120000080023120000080023120000080F23');
 MyStuffHex(ADR(icon[48]),
 '1200000BF923120000080F2312000008002311FFFFF00023');
 MyStuffHex(ADR(icon[60]),
 '08000000004307FFFFFFFFFA30100000000260FFFFFFFFFE2C')
 ;
 MyStuffHex(ADR(icon[72]),
 '18000000013832AAAAA8A9F0655555515380C2AAAAA82A58
 0');
 MyStuffHex(ADR(icon[84]),
 '800000000980FFFFFFFFF300800000001600FFFFFFFFFC00');
 END InitIcon;

```

```

PROCEDURE DrawIcon(h,v: INTEGER; mode : INTEGER);
VAR
 srcBits : BitMap;
 srcRect, dstRect : Rect;
BEGIN

```



```

srcBits.baseAddr := ADR(icon);
srcBits.rowBytes := 6;
SetRect(srcBits.bounds,0,0,48,32);
srcRect := srcBits.bounds;
dstRect := srcRect;
OffsetRect(dstRect,h,v);
CopyBits(srcBits,thePort^.portBits,
 srcRect,dstRect,mode,NIL);
END DrawIcon;

PROCEDURE MoveIcon;
BEGIN
 (* Erase the old icon *)
 DrawIcon(x[screen],y[screen],srcBic);
 (* Calculate new position *)
 x[screen] := x[1-screen] + xDir;
 (* Don't let icon go off screen *)
 IF (x[screen] < 0) OR (x[screen] > 460)
 THEN
 xDir := -xDir;
 END;
 y[screen] := y[1-screen] + yDir;
 IF (y[screen] < 0) OR (y[screen] > 310)
 THEN
 yDir := -yDir;
 END;
END;

```

```

 (* Draw the icon *)
 DrawIcon(x[screen],y[screen],srcOr);
 (* Swap our screen reminder *)
 screen := 1 - screen;
END MoveIcon;

BEGIN
 HideCursor;
 (* Re-launch ourself with alt. screen *)
 LaunchWithTwoScreens;
 InitIcon;
 Write(14c); (* Clear Screen *)
 CopyScreens;
 screen := 0;
 x[0] := 100;
 y[0] := 100;
 x[1] := 100;
 y[1] := 100;
 xDir := 1;
 yDir := 1;
 WHILE NOT Button() DO
 MoveIcon;
 SwitchScreens;
 END;
END Animate.

```



# Modula Mods

## Using Macsbug To Debug Modula-2 Programs



Tom Taylor  
MacTutor Contributing Editor  
MacTutor Vol. 1 No. 13

Developing a Mac application is not an exercise in trivial programming. With four or five hundred toolbox routines at the programmer's disposal, writing a Mac-like program can be somewhat frustrating. With no debugger available, the MacModula-2 programmer is usually baffled and angry when his program bombs with ID=02 or some other bomb box.

Although MacModula-2 does not come with a debugger, a machine-language level debugger by Apple can be used by the Modula-2 programmer. MacModula-2 traps most runtime errors such as range violation, integer, cardinal, real, and storage overflow, among others. The toolbox, however, may not be so forgiving. Passing bad parameters or calling routines in the wrong order can cause the infamous bomb box to appear. Fortunately, Apple's debugger can be very useful when debugging a Modula-2 program riddled with toolbox calls.

### MacsBug

In order to debug a program at the machine-code level, it is necessary to obtain a debugger. Apple supplies a number of debuggers with its *Macintosh 68000 Development System* package:

- *Maxbug* - a full-screen debugger for 512k Macs.
- *Macsbug* or *Midibug* - an 8-line debugger for 128k Macs.
- *Termbug A* and *Termbug B* - debuggers that display their information on an external terminal plugged into the modem or printer port.

I use Maxbug because of its 40-line display and the fact that it is the only debugger (of the ones mentioned) that actually displays the toolbox trap names during operation. The terminal-based debuggers come in handy once in a while because they allow the user to print the heap on the ImageWriter. This article and its examples will use Maxbug. Any one of the debuggers mentioned can be found in the following places:

- 1) Apple's *Macintosh 68000 Development System* package.
- 2) CompuServe. The debuggers have been uploaded to the Macintosh Developer's section.
- 3) Someone who has Apple's Software Supplement.

It would also be extremely helpful to obtain and read a copy of the documentation that describes the MacsBug debuggers. Before installing the debugger, you should install the "Programmer's Switch" that came with your Mac. This switch installs in the lower-left-hand side of the Mac. The

button closest to the front reboots the Mac and the rear button generates a non-maskable interrupt. Pushing this button without the debugger installed generates a bomb box with ID=13. With the debugger installed, it simply places you in control of the debugger. Regardless of which debugger you wish to use, you must copy it to your boot disk and name it *Macsbug*. Whenever you boot with this disk, the debugger will automatically be installed (note: this means that after copying the debugger to the boot disk and renaming it, you must shutdown and re-boot in order to install the debugger). If you have not installed your own startup screen on your boot disk, you'll see a "MacsBug Installed." message under the "Welcome to Macintosh" bootup message (see figure 1). If you have installed your own startup screen, no message will be displayed but the debugger will still be installed.



Figure 1. Startup screen with MacsBug on the boot disk.

If this is your first experience with the debugger, wait till the Finder has come up and the disk drives have stopped spinning and hit the debug button on the side of the Mac. If you are using Maxbug, the screen should clear and display something like figure 2. If you are using Midibug, you'll see figure 2 on the bottom part of the screen. Finally, if you are using one of the terminal debuggers, you'll see figure 2 on the external terminal. Generally, whenever you enter the debugger, all of the registers are displayed and the next line to be executed is disassembled. The debugger, à la conventional (i.e. pre-Mac) programs, displays a ">" prompt and waits for the user to enter a debug command. There are many commands that can be entered here. For starters, though, enter a "G" (in these debug examples, the bold type highlights the commands that should be typed by you, the user). The screen will display the desktop again and the Mac will resume normal operation.

```

404FA6: SUBQ.B #54,D1
PC=00404FA6SR=00002004TM=000035EA
D0=00000020D1=0001FFE0D2=A0000000 D3=FFFFFFE1
D4=0000FFFFD5=00000069D6=00000001 D7=0000FFFD
A0=200219FCA1=0000E85AA2=A0019F3A A3=A0019EEC
A4=0000ECD6A5=0007771CA6=00076CB8 A7=00076C8A
>

```

**Figure 2. 1-Disassembled listing of next line to be executed, 2-Current stack pointer, and 3-Debugger prompt.**

There are at least four ways to enter the debugger:

- Assemble the debugger trap (\$A9FF) into an assembly language program.
- Perform some operation that generates a Macintosh serious error that would normally put up a bomb box.
- Press the interrupt button on the side of the Mac.
- Enter commands to the debugger that cause a program to enter the debugger when certain conditions are met.

Since the debugger "catches" the Mac system errors (at least most of them!), I almost always have the debugger installed. This allows me to continue operation without having to reboot. More on this later... Some programs will not work with the debugger installed. These programs include those that use the alternate screen buffer. Apparently the debugger and the alternate screen live in the same area in memory. Since the debuggers chew up a lot memory, some big programs won't run on a 128k Mac. The MDS Edit program, for example, hangs on a 128k Mac with the debugger installed. Sometimes I want as much memory as possible in Switcher so I remove the debugger by renaming it, and rebooting.

### A Debug Example

When a MacModula-2 program dies with some system error, I usually perform the following operation in order to narrow down the location of the bug. The debugger is capable of printing out the name of each toolbox routine whenever the routine is called. Furthermore, the debugger can limit the printing of the toolbox names called from a certain range of addresses. This is an important feature. It allows us to only see the toolbox calls made from our Modula-2 program and not calls made by the toolbox itself. From now on, I'll call a "toolbox call" a trap or A-trap. Toolbox calls are not called directly as subroutines. Instead, each toolbox routine is assigned a 16-bit value where the first four bits are 1010 (or a hex A, hence the name, A-trap!). No legal 68000 instruction begins with the 1010 bit pattern. Whenever the 68000 processor hits one of these words that begin with 1010, it

generates a '1010' interrupt and traps the interrupt to a specific address. At that address there is a routine that uses the last 12 bits of the trap to determine whether the trap is a Toolbox trap (parameters passed on the stack) or an O.S. trap (parameters passed in registers) and uses 8 of the bits as an index into a trap dispatch table. By using this technique, Apple has cleverly extended the 68000 instruction set and made a way to patch various routines easily. The debugger can perform many operations with the different traps, some of which will be discussed later.

The most common error on the Mac seems to be the address error (ID=02). This occurs whenever the 68000 tries to perform some word or longword operation with an odd-address. The second most common error is an illegal instruction (ID=03). Generally when you get this error, your program has jumped off somewhere in memory and attempted to execute data or garbage. In order to try out the debugger, let's create a simple Modula-2 program (figure 3) that generates an address error and examine what happens when the program is executed.

```

MODULE AddressCrasher;
TYPE
 Ptr = POINTER TO CARDINAL;
VAR
 ptr : Ptr;
 card : CARDINAL;
BEGIN
 ptr:=NIL; (* Set ptr to 0 *)
 INC(ptr); (* Make ptr odd *)
 card := ptr^; (* Crash because we
 try to access a
 word on an odd
 address boundary *)
END AddressCrasher.

```

**Figure 3. A Modula-2 program to generate a Macintosh address error**

When the program in figure 3 is executed, it will generate an address error and invoke the debugger and display something like the first part of figure 4 (everything up to the first ">" prompt). The Modula-2 program crashes because it sets up a pointer equal to 1 and tries to access a word variable at that location (remember: accessing a word at an odd address is a no-no on the 68000). Figure 4, arrow 1, shows how the debugger prints the offending address (00000001). By examining the dumped registers, we can generally determine which register held the bad address (see arrow 2). The instruction that caused the address error is usually in the immediate vicinity of the current PC. By using the debugger's disassemble command (IL), I start disassembling the code 10 bytes in front of the PC (see arrow 3). That gives a nice window of instructions from where the error occurred. Notice how "PC" is displayed right in front of one of the instructions. That instruction is the next one to be executed.

The instruction immediately preceding the "PC" is the instruction that caused the address error (see arrow 4). The instruction tried to access the contents of the word of A0, or 00000001, an odd-address. To get back to the finder without rebooting, simply use the Exit-to-Shell (ES) debugger command.

```

ADDR ER 00000001
00E5EA: JMP *-$0C9C ; 0000D94E
PC=0000E5EASR=00002000TM=00003A19
D0=00000C8AD1=00012162D2=0000FFFF D3=00010000
D4=0000E01AD5=00011E98D6=00012120 D7=00000060
A0=00000001A1=0001213AA2=00012156 A3=000120F0
A4=00012162A5=0000D95CA6=0000E0F0 A7=000775C4
>IL PC-10
00E5DA: JMP *-$0C8C ; 0000D94E
00E5DE: MOVE.W (A6)+, $001E(A3)
00E5E2: JMP *-$0C94 ; 0000D94E
00E5E6: MOVEA.L (A6)+, A0
00E5E8: MOVE.W (A0), -(A6)
00E5EA: PC JMP *-$0C9C ; 0000D94E
00E5EE: MOVEA.L (A6)+, A0
00E5F0: ADDQ.L #2, A0
00E5F2: MOVE.W (A0), -(A6)
00E5F4: JMP *-$0CA6 ; 0000D94E
00E5F8: MOVEA.L (A6)+, A0
00E5FA: ADDQ.L #4, A0
00E5FC: MOVE.W (A0), -(A6)
00E5FE: JMP *-$0CB0 ; 0000D94E
00E602: MOVEA.L (A6)+, A0
00E604: ADDQ.L #6, A0
00E606: MOVE.W (A0), -(A6)
00E608: JMP *-$0CBA ; 0000D94E
00E60C: MOVEA.L (A6)+, A0
00E60E: ADDQ.L #8, A0
>ES

```

Figure 4. Debug display of Figure 3's execution

### MacModula-2 Internals

Before moving on to a more complicated and more realistic debugging example, a few words on the internals of the MacModula-2 interpreter would be useful. As you are probably aware, the MacModula-2 compiler does not generate 68000 native code. Instead, the compiler generates an intermediate code called "M-Code." The Modula-2 program is responsible for the following:

- Loading .LOD files and program overlays from disk into memory.
- Interpreting each M-Code by executing a number of 68000 instructions that represent the function of the M-Code.
- Providing the necessary interface between Modula-2 and the Mac's ToolBox.
- Provide terminal handling routines for the Terminal and InOut modules.

This article deals primarily with the interface between Modula-2 and the ToolBox. In MacModula-2, there are two different ways to call ToolBox routines. The first, and cleaner of the two ways, is to simply import the desired ToolBox routine into a program. When the M2 Linker links a program that makes ToolBox calls, it generates a CX (Call External) M-Code to a hardwired module and procedure number (there is a specific module number for every ToolBox manager and a distinct procedure number for every procedure in each manager). The second method of calling ToolBox routines involves "cutting and pasting" ToolBox code procedures into a source module. Each of the code procedure has the module and procedure numbers hardcoded right into the routine. These numbers are the same that the M2 Linker assigns while linking modules using the import method of calling ToolBox routines. In either case, when the Modula-2 interpreter executes a CX M-Code and the module number belongs to a ToolBox manager, a number of table lookups are made and the interpreter jumps to an appropriate glue routine. There is a glue routine for every ToolBox trap supported by MacModula-2. The glue routine is responsible for moving parameters off of the Modula-2 expression stack and putting them on the Mac's hardware stack (or in registers if the trap is an O.S. trap) and calling the correct ToolBox trap. If the ToolBox trap is a function, then the glue routine must pop the result off of the Mac's stack and push it on the Modula-2 stack. I mention this because it will help you to understand any disassembled code immediately surrounding most traps found in the Modula-2 interpreter.

This next example shows how to trace the various ToolBox calls made from Modula-2 and help isolate the source of errors. The following example is a simple program that puts up the Apple menu and allows the use of desk accessories (see figure 5). This program has a built-in bug and the goal of this example is to find the bug. After launching the program and opening a desk accessory, clicking inside the accessory on the desktop causes the program to die with an address error. Here are the steps necessary to debug this program:

- (1) Execute the program in figure 5 and open a desk accessory (the calculator, for example).
- (2) Position the mouse cursor over the calculator's close box and press the interrupt button on the side of the Mac.
- (3) At this point, we want to tell the debugger to display any Mac A-trap called from the Modula-2 interpreter. In order to tell the debugger to only display traps called from the interpreter, it is necessary to give the debugger the starting and ending addresses of the interpreter's code in memory. The Modula-2 interpreter is written as one code segment. By typing HD 'CODE', the debugger will display heap information about the code (see figure 6).
- (4) The HD 'CODE' command will cause the debugger to print out three lines. We are interested in the middle line. The third field on the middle line is the length (in hex bytes) of the Modula-2 interpreter's code. The fifth field contains an address of a pointer that points to four bytes before the

beginning of the code. The actual address of the start of the code is found by using the DM (display memory) command. Typing the '@' character in front of an address or register causes the debugger to treat the address or register as a pointer and go indirect through that address or register. In this case, I typed: DM @CC18+4. This means, "Get the longword contents at the address CC18 (which happens to be CE8E), add 4 to it, and display 16 bytes of memory at that location." The calculated address in this example, CE92, is the beginning of the Modula-2 interpreter's code.

(5) The next step is to use the debugger's AT (A-trap Trace) command to tell the debugger to trace all traps called from the Modula-2 interpreter. The AT command takes up to four parameters. These parameters are: starting trap number, ending trap number, starting address, and ending address. All traps with a number between the starting trap number and the ending trap number called within the starting and ending address will be displayed.

(6) Execution is continued by typing G. Using Maxbug, the screen will really flicker while the debugger traces the traps and switches between the normal and debugger screens. Since we are trying to debug the situation when the mouse is pressed inside a desk accessory, go ahead and press the mouse (and hold it until the program bombs with an address error). Look at the address of the disassembled line immediately after the address error message. If the address is in the 400000 range, then the program died in a ROM ToolBox routine. Look at the last trap printed out. This routine is the one suspected of causing the crash. Those using a debugger other than Maxbug will only see trap numbers and not trap names. You'll need to look up the trap name in one of the many trap lists that exist (like the one in the back of *Inside Macintosh*).

(7) At this point, it's time to look back at the Modula-2 source code (figure 5) and examine the parameters to the SystemClick routine. After a quick peek, it's obvious that a bad parameter was passed to the routine.

### Modula-2 ToolBox Debugging Example

```
MODULE ToolBoxCrash;
```

```
(* This program simply allows the
 use of desk accessories. This
 program will probably bomb in the
 SystemClick() routine at the end
 of the program. *)
```

```
FROM MenuManager IMPORT
 menuHandle, NewMenu,
 AddResMenu, InsertMenu,
 DrawMenuBar, MenuSelect,
 GetItem, AppendMenu,
 HiLiteMenu;
```

```
FROM MacSystemTypes IMPORT
 Str255, LongCard;
```

```
FROM Strings IMPORT
 StrModToMac;
```

```
FROM DeskManager IMPORT
 SystemTask, SystemClick,
 OpenDeskAcc;
```

```
FROM EventManager IMPORT
 EventRecord, GetNextEvent;
```

```
FROM WindowManager IMPORT
 FindWindow, WindowPtr;
```

```
VAR
 menu : ARRAY [0..2] OF menuHandle;
```

```
PROCEDURE SetUpDeskAccs;
```

```
 VAR
 appleHeader : Str255;
 BEGIN
 appleHeader[0] := 1c;
 appleHeader[1] := CHR(14h); (* apple mark *)
 menu[0] := NewMenu(1,appleHeader);
 AddResMenu(menu[0],'DRVr'); (* add desk acc's *)
```

```
 StrModToMac(appleHeader,'Exit');
 menu[1] := NewMenu(2,appleHeader);
 StrModToMac(appleHeader,'Quit');
 AppendMenu(menu[1],appleHeader);
```

```
 InsertMenu(menu[0],0);
 InsertMenu(menu[1],0);
 DrawMenuBar;
 END SetUpDeskAccs;
```

```
VAR
 event : EventRecord;
 window, badWindow : WindowPtr;
 menuID : LongCard;
 accessory : Str255;
 refNum : INTEGER;
```

```
CONST
 everyEvent = -1;
 mouseDown = 1;
 inMenuBar = 1;
 inSysWindow = 2;
```

```
BEGIN
 SetUpDeskAccs;
 LOOP
 SystemTask;
 IF GetNextEvent(everyEvent,event) THEN
 IF event.what = mouseDown THEN
 CASE FindWindow(event.where>window) OF
 inMenuBar:
 menuID.r := MenuSelect(event.where);
 IF menuID.h = 1 (* Apple Menu *) THEN
 GetItem(menu[0],menuID.l,accessory);
 refNum := OpenDeskAcc(accessory);
 ELSIF menuID.h = 2 (* Quit Menu *) THEN
 IF menuID.l = 1 THEN EXIT END;
```

```

END;
HiLiteMenu(0);
| inSysWindow:
(* Pass a bad window ptr here! Change
 badWindow to window to make the program
 work correctly.
*)
SystemClick(event,badWindow);
ELSE
END;
END;
END;
END;
END;
END ToolboxCrash.

```

**Figure 5 (below): Tracing a Modula-2 program's ToolBox calls**

```

078B60: MOVE #$2700,SR
PC=00078B60SR=00002004TM=000018E0
D0=00000000D1=00000000D2=FFFF0000 D3=00000000
D4=00000014D5=00000000 0000
A0=00015BB2A1=0040BB 8DA
A4=0000E896A5=0000778 57E
>RD 'CODE'
0000CB00
0000CE86 H 000048D0 E 0000CC18 * 20 0001 CODE
CNT ### 0001 000048D0
>DM @CC18+4
00CE92 4E56 0000 2038 0330 6652 B0B8 0328 664C NV.. 8.0fR...(fL
>AT 0 1000 CE92 CE92+48D0
>G
MACTRAP A9B4 SYSTEMTASKPC:00010648 A0:0000E0F0D0:00001984 TM:000018F7
MACTRAP A970 GETNEXTEVEPC:0000FF70 A0:0000E0EAD0:00001984 TM:000018F8
MACTRAP A9B4 SYSTEMTASKPC:00010648 A0:0000E0F0D0:00001984 TM:000018F7
MACTRAP A970 GETNEXTEVEPC:0000FF70 A0:0000E0EAD0:00001984 TM:000018F8
MACTRAP A9B4 SYSTEMTASKPC:00010648 A0:0000E0F0D0:00001984 TM:000018F7
MACTRAP A970 GETNEXTEVEPC:0000FF70 A0:0000E0EAD0:00001984 TM:000018F8
MACTRAP A92CFINDWINDOWPC:00010638 A0:0000E0E8D0:00001984 TM:000018FE
MACTRAP A9B3 SYSTEMCLICPC:00010638 A0:0000E0E8D0:00001984 TM:00001900
ADDR ERRFFFFFFFFF
40DFDC: BNE.S *+$0020 ; 0040DFFC
PC=0040DFDCSR=00002000TM=00001901
D0=000059B4D1=0000088CD2=0000001C D3=0000FFFF
D4=00000014D5=0000001CD6=0000FFFF D7=00000000
A0=000775A8A1=FFFFFFFFFA2=0000FDA8 A3=00011AEC
A4=0000E896A5=00077844A6=000775A0 A7=00077542
>ES Although this method of tracing and isolating bugs may
seem difficult, I can assure you that it is worthwhile and much

```

simpler and faster than peppering a program with "Print" debug statements and recompiling. In fact, I have debugged many, many Modula-2 programs that people have sent me without ever looking at the source code. Even though this article has only brushed the surface of debugging techniques and how to use the debugger, hopefully it will give the MacModula-2 user enough confidence to try some low-level debugging and gain a better understanding on how the Mac works.



# APL Adventures

## A Beginner's Look at Mac APL

Allyn Weaks

Seattle, WA

MacTutor Vol. 1 No. 12

I've wanted to learn APL ever since I heard that it could do integrals in one line and matrix inversions in a single symbol. Until PortaAPL for the Mac came along, I never had access to a computer with a full implementation. I'm hardly an expert at this language, having played with it for all of a month now, so please forgive any minor blunders. This month I'll start with a description of APL, enough of a tutorial to get you started defining simple functions (including an overview of the syntax, data types, and some of the simpler functions), and a brief review of PortaAPL for the Mac. I'll spend as few articles as possible in straight APL tutorial mode before starting in on Mac-specific features and bigger programs.

APL, A Programming Language, was developed by Kenneth Iverson at IBM in the early 60's, which makes it one of the older computer languages. It is more of a mathematical notation that can be interpreted by a computer than a traditional computer language, and handles data in groups (arrays) rather than just in small pieces.

Though it's a natural language for scientific and mathematical problems, in the U.S. it seems to be used mostly as a business language. Almost all of the textbooks currently in print are slanted toward business. Insurance companies in particular like it, because APL is ideal for statistical programs. In Europe and Canada it caught on more quickly and is used a lot in the scientific community for data analysis.

Since it is an interpreter, APL runs slower than most compilers. But it runs much faster than other interpreters such as Basic. The programs are much shorter, so APL spends most of its time working on the problem, not interpreting the commands. You also get the advantages of an interpreter - interactive writing and debugging. Functions are defined by name as in Forth or Logo, and you can define local variables, so it's easy to define new functions as you go along.

APL is extremely powerful. It's recursive and self-modifying, with dynamic allocation of data storage. It does calculations on entire arrays at once, not just element by element. This power has its disadvantages, the primary one being memory limitations. Arrays require lots of memory. On a 512k Mac, the largest two dimensional array of integers you can deal with is about 225 by 225, which isn't big enough for many problems. Sometimes you have to use a less elegant algorithm to solve a problem. But then, who has ever had enough memory?

The peculiar character set APL uses (fl, @, ", %, Å, etc.) has given it the reputation of being unreadable. Pascal programmers have been heard to say that APL programmers

hand each other sections of code and say 'Bet you can't figure out what this does!'. But APL isn't really harder to read than other languages once you know what the symbols mean. And since APL can work on groups of data all at once, instead of just a little at time, the code is much more compact. You can see the whole thing at once, instead of getting lost manipulating indices in the middle of triply nested loops.

APL is very different from Fortran, C, Forth, or Lisp. The symbols, syntax, and natural algorithms all take some getting used to. The first hurdle is the keyboard layout. Unshifted characters give upper-case letters, shifted characters and numbers give the APL symbols. You'll need to keep the keyboard map handy for reference at first, though most of the symbols are easy to find once you know the mnemonics (i.e. Ë (iota) is shift-I, ® (rho) is shift-R). If you can't stand it, there is an ASCII mode that substitutes keywords and ASCII punctuation for the symbols, but if you plan to read standard APL books, or trade programs with other APL users, it's better to do it right. The real notation is also more compact and easily read - the symbols stand out.

The next difficulty is the syntax. There is no hierarchy of operators - all APL expressions are evaluated from right to left. If you need to change the order of operation, you must use parentheses. For example, the expression 5-6-2 is equivalent to 5-(6-2) = 1 in APL, not (5-6)-2 = -3 as in most languages. And 5x6-2 equals 20, not 28.

The terminology is a bit different too. What is usually called an operator, such as + or -, is a function in APL, because it returns a value. An operator is a special thing that systematically modifies the effect of a function. Functions operate on data, and operators operate on functions. Operators are one of the things that makes APL so powerful - many mathematical ideas such as the inner product (given two vectors a,b,c and x,y,z the standard inner product is ax+by+cz, a scalar) can be generalized to any functions, not just the sum of the products.

Functions can be niladic, monadic, or dyadic. Niladic functions take no arguments - some system commands such as )OFF are niladic. Monadic functions take one argument, and are called by the sequence FUNCTION ARGUMENT, such as the factorial of N (!N). Dyadic functions, which take two arguments, are called as ARGUMENT FUNCTION ARGUMENT, as in 5 + 3. This is true of user defined functions as well as the built in primitives.

People who insist on strongly typed languages can stop here and read the Pascal article instead. The standard data types are boolean, character, integer, and floating point, but these distinctions are less important than the structure of the data into scalars, vectors, and larger arrays. An argument to any

function can be a scalar, vector, or array, with a few exceptions. It is irrelevant (again with some exceptions) whether the data is a character string, an integer, or a set of boolean values. A scalar is a single number or character, without dimension or length. A vector is a set of data that forms a one dimensional array, a matrix has two dimensions. Arrays can have up to 63 dimensions in PortaAPL, though you'll never use that many - if you had two bits per dimension, you'd need a billion Gigabytes to hold the array. All elements of an array must have the same data type, so if you put a single floating point number in a large array, every element will be floating point. Type conversions are taken care of automatically and transparently at the interpreter's discretion, though there are ways to force a conversion to a particular type. On the Mac, boolean values are stored as bits, characters as bytes, integers as 4 byte words, and floating point as 8 byte long words. A character string is a vector of characters; a character matrix is a set of strings, one per row.

In addition to dividing functions into classes according to how many arguments they take, they can be divided into three groups that depend on what their effect is. Scalar functions change the data in an array, but not the shape or size of the array. Restructuring functions change the shape or dimension of an array, but not the data values. Mixed functions change both the data and the shape, and will have to wait for a later article. A scalar function can operate on arrays, not just scalars, but they operate element by element. For example, you can add two vectors of the same length:

```
2 4 6 11 8 + 7 43 1 6 31
9 47 7 17 39
```

Iota (È) and rho (®) are two of the most important reshaping functions. Iota in it's monadic form creates a vector of integers in ascending order from a scalar: È6 gives the vector 1 2 3 4 5 6.

Monadic rho gives the length of each dimension of an array: ® 1 3 5 7 is 4. Dyadic rho creates an array with the dimensions given by the left argument, filled with the values given in the right argument. The right argument is repeated as many times as necessary to fill the array:

```
2 4 ® 1 2 3
```

```
1 2 3 1
```

```
2 3 1 2
```

Some of the scalar functions

will look deceptively familiar to programmers. + and - do what you expect, but \* means the exponential or power, not multiply, and / is not divide, but something more complicated - it can be either a restructuring function (compress) or an operator (reduction). When used as the reduction operator, f/ inserts the function f between each element of a vector:

```
+ / 1 2 3 4
10

* / 1 0 0 1
1

- / 1 2 3 4
TM2
```

There is a floating point benchmark that's become popular on usenet: how long it takes to calculate the harmonic series to 10000. This is the sum of 1/i from i=1 to 10000. Most languages take 5 or so lines to write this program; APL takes 9 characters, including the 10000. No looping is needed.

Start by noticing that you want to do something with an index that ranges from 1 to 10000. Well, there's a function iota that will generate a vector with all those values. Then, you need to take the reciprocal of each value, so just use the monadic ±. To take the sum of the vector, use the reduction

### Summary of the Scaler Functions

|    |                | Monadic      |                       | Dyadic  |
|----|----------------|--------------|-----------------------|---------|
| -  | identity       | 3 " -3       | addition              | 8 " 5-3 |
| -  | negation       | 3 " —3       | subtraction           | 4 " 6-2 |
| ≠  | signum         | TM1 " TM6    | multiply              | 6 " 3*2 |
|    |                | 0 " 0        |                       |         |
|    |                | 1 " 6        |                       |         |
| ±  | reciprocal     | .2 " ±5      | divide                | 2 " 6±3 |
| π  | exp(a)         | 2.7183 " π1  | power                 | 8 " 2π3 |
| °  | Naperian log   | 2.3026 " °10 | log to base A of B    | 2 " 3°9 |
| β  | round up       | 5 " β4.1     | maximum               | 4 " 4β3 |
| ð  | round down     | 5 " ð5.9     | minimum               | 3 " 4ð3 |
| μ  | absolute value | 6 " μTM6     | modulo                | 1 " 4μ9 |
| !  | factorial      | 6 " !3       | binomial coefficient  |         |
|    |                | 6 " 2!4      |                       |         |
| øø | pi times A     | 6.2831 " øø2 |                       |         |
| ø  | random         | 3 " ø5       |                       |         |
|    |                | 5 " ø5       |                       |         |
| †  | not            | 1 " †0       |                       |         |
| ∧  |                |              | and                   | 1 " 1∧1 |
| ∨  |                |              | or                    | 1 " 1∨0 |
| ∨  |                |              | nand                  | 0 " 1∨1 |
| ∨  |                |              | nor                   | 0 " 1,0 |
| £  |                |              | less than             | 1 " 3£5 |
| ¢  |                |              | less than or equal    | 0 " 4¢3 |
| ∞  |                |              | equal                 | 1 " 5∞5 |
| §  |                |              | greater than or equal |         |
|    |                | 1 " 4§2      |                       |         |
| ¶  |                |              | greater than          | 0 " 2¶4 |
| .  |                |              | not equal             | 0 " 5.5 |

Fig. 1 APL Table of Functions



operator with addition. So the whole thing turns into  $\rightarrow \pm 10000$ . How long does it take? 49 seconds, compared to 38 seconds for Aztec C, and 69 seconds for Megamax C. APL won't always do this well compared to compilers, this set of functions is extremely efficient in APL. Notice also the amount of memory used for a vector of 10000 4-byte integers.

To turn this into a function so you can run it for any number of terms, enter the editor by typing `© Z " HARMONIC N <ret>`. This line will appear as line zero at the top of the screen. Any variable appearing on line zero is a local variable, all others are global. If you don't assign the function to a dummy variable (Z in this case), you won't be able to call it from another function later. Next, starting with line one, type in your program. At some point, the final value of the program must be assigned to the same dummy variable as in line zero. When you're finished, the screen should look like this:

```
©Z " HARMONIC N
[1] Z " $\rightarrow \pm \text{EN}$
©
```

You can move the cursor with the mouse. The lamp symbol (Ê) can be used at the beginning of a line to put in comments. Use command-z or the EDIT menu to exit the editor. To define a dyadic function, put one argument in front of the function name, and one after. DICE will roll N dice with M sides each:

```
©Z " N DICE M; A; B
[1] Ê D&D dice roller -
 rolls N M-sided dice
[2] A " N ® M
[3] B " œA
[4] Z " $\rightarrow B$
[5] Ê This can also be written on
 one line as:
[6] Ê Z " $\rightarrow \text{œN} ® M$ or Z " $\rightarrow B " œA " N ® M$
©
```

Line 2 sets up a vector with the same number of elements as there are dice and with each element equal to the numbers of sides of the die. The scalar operator ?R chooses a random element of the vector (ÊR), and if you give ? a vector as an argument it will find the random number for each element. Line 3 just adds up all of the independent rolls. What sort of changes need to be made to roll several dice with different numbers of sides? None! M can be a vector containing the sides of each die. Make sure the number of elements agrees with the value of N, however, or you will get the wrong result. There's at least one way to take care of this without changing the function, and I'll leave the finding of it as a puzzle.

In addition to the built in functions, APL has a number of system functions and system commands that are fairly standard from one implementation to another. System

commands are prefaced with a right parenthesis, such as `)OFF` to exit to the finder, `)SAVE FILENAME` to save the workspace, and `)COPY FILENAME` to add a workspace to the current workspace. System functions, or quad functions, start with the quad symbol  $\rightarrow$ . These can be used to inquire about or set various system variables, such as the precision of results with  `$\rightarrow$ PP`, or to manipulate files. There are also quad versions of  `$\rightarrow$ OFF`,  `$\rightarrow$ COPY` and some other commands so that they can be called from inside of functions.

PortaAPL is written and distributed by Portable Software in Cambridge MA (60 Aberdeen Avenue, Cambridge, MA 02138; (617) 547-2918). A 512k Mac is required. The latest version is 2.0m. The price is \$275 for the interpreter, some very clear documentation, and several workspaces with sample programs and functions to access parts of the toolbox. Upgrades are available for \$25. Best of all, the interpreter isn't copy protected, so you can run it easily from a hard disk.

Portable Software has been writing APL interpreters for several years and for several different computers including Vax and the IBM PC. PortaAPL is a full IBM mainframe implementation, with lots of nice extensions. In fact, the main reference manual provided is the official IBM APL manual. The differences to the IBM version are an improved screen oriented editor, and the workspace id format which is system dependent. (Workspace id is a fancy term for program file name.) Extensions include ASCII mode, access to the Mac file system (though not the resource fork), and a way to write assembly language functions. The keyboard is a standard APL keyboard, with extensions for the Mac. Overstruck characters can be typed with the option key instead of key1-backspace-key2.

Another nice feature is a built in Datamedia 1520 APL terminal emulator. If you have access to APL on another computer, you can call up and work from your Mac. The Mac allows an improved keyboard set up, so you can send lowercase characters if you need to by holding down the option key. Unfortunately, there is no file transfer, so you can't send programs back and forth with it.

The program is written in C so it isn't as compact or as fast as it could be. It's 180k, and allows a work area of about 221k. Integers are 4 bytes and floating points numbers are 8 bytes with a range of  $\pm 1.79E308$ . Floating point calculations use the SANE package. The largest allowed function is 9999 lines; the maximum length of a symbol is 77 characters and underscores are allowed in symbol names. 16 files can be opened at once.

Editing functions and character matrices is acceptable, and certainly an improvement over the standard APL editor. (Try reading through some of the descriptions of editing a function in one of the books listed at the end!) The cursor is moved with the mouse, and there are menu and keyboard commands to insert and delete lines. Because of the nice fonts on the Mac, overstruck characters are typed in with the option key, so the editor can be in insert mode all of the time. Cut and paste aren't available, which can be a nuisance.

An excellent DRILL program is provided, that randomly generates APL expressions at 5 levels for you to evaluate. Since it evaluates the answer you give it and compares that to the evaluation of the problem, you can add to the challenge by coming up with expressions that are equivalent, but use different functions.

One of the workspaces that comes with the package is called Goodies. It has lots of useful stuff in it, including peek and poke, a sound function for playing simple tunes, mouse functions, printer i/o redirection, and functions to let you access the modem port. There are also routines that copy text or graphics from the clipboard to the screen, or text from the screen to the clipboard.

As of 2.0m, PortaAPL supports much of quickdraw and menus, but not windows, resources, or controls. Desk accessories are available from inside the interpreter. Although the EDIT menu doesn't show the standard cut, copy, and paste functions, command-X, and V are functional when desk accessories are active. The calls to quickdraw and the menu manager are straight-forward. To use them, you need to include the appropriate workspace. The quickdraw workspace includes functions for line drawing, rectangles, ovals, arcs, text fonts and scroll regions. Not all of quickdraw is supported - getpixel, regions, pictures, copybits, and icons are all missing. The lack of getpixel is going to make it much harder for me to do screen dumps to my non-Imagewriter printer. Menus are easily created by assigning a character matrix containing the items and the functions they run to a menu id number, then installing the menu. The interrupts are handled by APL if you are at the command level. From inside

of programs, there is a function —GETKEY that lets you ask for the appropriate events. Windows aren't supported, but you can define regions for input and output.

All in all, PortaAPL is a nice job. I haven't found any bugs yet while putting in sample programs from various sources. There is one thing to beware of - you must remember to save your work before you exit, because APL won't remind you. It would be nice to have complete access to the toolbox, but that will come eventually - if not from Portable Software, then from anyone who is willing to do a bit of assembly language programming. It's expensive compared to Basic, but it's much more fun.

I hope this has been enough to get you started. I recommend that you buy or borrow an APL text book and work through plenty of examples, as well as the Drill program. Next month, I'll get into loops and branches, the other operators, and a few ways to generate prime numbers.

A Programming Language by Kenneth E. Iverson  
John Wiley & Sons 1962

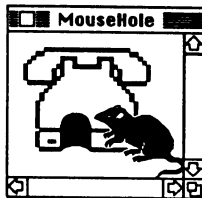
An Introduction to APL by S. Pommier  
Cambridge University Press 1984

APL: An Interactive Approach by Gilman & Allen J. Rose  
John Wiley & Sons 1983

Structured Programming in APL by D. Geller & D. Freedman  
Winthrop 1976



# Mousehole Reports



*Rusty Hodge  
SysOp & Contributing Editor  
MacTutor Vol. 1 No. 2*

One thing that was very nice to see was a "almost" release version of Microsoft Word. It was fast, and I mean fast!  
- Mac Scotty

The BIG news is from a company called General Computer. They announced a Mac mod called HyperDrive, which is a RAM expansion to 512K, and the installation of a 10 meg hard disk with the controller INSIDE THE MACINTOSH. This allows direct booting from the hard disk, free modem port, no serial I/O to slow things down, and no external box to carry around. Price is \$2,795 on a 128K

machine or \$2195 on a 512K machine. They do the installation or you can buy a kit from your dealer.

Jazz from Lotus is supposed to ship March 1985. It's excellent. Five applications can be open at once. Spreadsheets can be pasted to charts, which can be pasted to word processing. But the real killer is when the spreadsheet was updated, the chart in the word processor changed! That's the first time word processing has been dynamically linked to a spreadsheet.  
- Silver See



## Mousehole Report

*MacTutor Vol. 1 No. 3*

### Benchmarks

Here are some "Sieve" benchmark measurements in case no one else has provided them:

|                            |               |
|----------------------------|---------------|
| MacPascal .....            | 1,270 seconds |
| Basic 2.0 .....            | 1,156 seconds |
| Basic 2.0 (compressed).... | 1,040 seconds |

I tried programming MacBasic V.82 but had series problems with system errors. (It's not ready for anything serious yet). Interesting part was that it said "DIM Flags(8190) exceeded dimension limits". Congrats on issue two; it's stuffed with lots of goodies and will take a month to explore each of the different articles. I'm kinda disappointed that MacPascal was slower than Basic. The "compressor" aspect of Basic 2.0 looks like a good efficiency improver with 15-20 percent for this small program.  
JCOM

### MacTech (Early MacTutor)

Just got my first MacTech and it looks great!!! Finally something to explain that #\$\$\* @ IM manual! If you are considering sending in your \$\$ to get MacTech, do it now. Don't wait, you won't be sorry. What do I need to do to get Vol. 1, No. 1? I'll send my right arm to get that assembly shell. Sounds great.

THE ATOM

### MS Word

I just got the Dec. 15 version of MS Word and I must say that it is just what I need in the way of a real word processor. You can have four documents open at once and you can divide them into two sections just like Multiplan. I had heard that the older

versions of Word worked really slow, but this seems to work fine and has drivers for lots of letter quality printers. It is not all that easy to change fonts and stuff, but how many times do you really use more than 2 or 3 fonts in a file? Also you have a horizontal scroll bar that goes out to somewhere around 18", so that problem is history. Also support of footnotes and other goodies make this the real Word processor that anyone who needs series word processing needs. This version still doesn't have multiple rulers, but I will call them up and see if they are going to implement them. And as most MS products, Word has a nice help file.

THE WATCHMAN

### Wait For Call...Forever!

When using MacTerminal and Apple modems, do NOT select "Wait for Call". Just do nothing after starting and it WILL answer calls. With Hayes modems, DO select "Wait for Call". Apparently the Apple modem isn't as Hayes compatible as we thought.

ROBERT WIGGINS

### Software Supplement Worth It!

If anyone is considering buying INSIDE MACINTOSH from Apple, spend the extra \$100 and get the software supplement too. Not only are you the first on your block to get the new Mac utilities (like resource editor), but you also get all the Lisa software. Useless if you don't have a lisa you ask? The software is no good to you, but I've already gotten my money back by using those disks instead of buying another couple of boxes of blank ones!

BURRILL SMITH



*The mousehole is a private Mac BBS, home to most of MacTutor's editorial board. Membership may be obtained by invitation of the Sys Op. Address membership requests to The Mousehole care of this publication. Include name, address and phone and the handle you wish to use. At present, there is no charge for membership.*

## **Helix Looks Good**

A note about Helix. Everyday I work with it, it gets better and better, especially the ability to change reports and how the data fields are oriented on paper. Supports a wide range of forms from labels to 15" imagewriter printer paper.  
-The Terminator

## **\$600 Upgrades?**

Yesterday, the New York Times reported the price of 512K Macs had dropped \$400. Also I called Apple and they said upgrades to a 512K Mac are now \$600. At the Apple Stock Holder's meeting yesterday, Andy Herzfeld demonstrated a program he wrote called "Switcher". It allows you to run up to 4 programs on a 512K Mac at the same time, switching contexts between them like slides on a projector. WOW!  
-Katz

## **Apple Irvine Ripped Off**

Apple Irvine had about 100 machines heisted last week (Iic, Iie, 128K's, 512K's). An Apple dealer can give you the serial numbers if someone wants to check out a deal too good to be true.  
- MACGEORGE

## **\$700 Upgrade?**

Yup. It's official. 512K RAM - \$700.00!

-Macowaco

## **MacWrite 3.95 Bug**

I've just discovered a VERY annoying problem with MacWrite 3.95. In case you haven't heard, MacWrite has a new upper limit on the number of paragraphs that it will allow within a document. Well, paragraphs are nothing more than carriage returns, and most downloaded ASCII text files contain carriage returns at the end of each line. Tonight I downloaded all the new messages from the mousehole and got a 23K ASCII file. When I tried to open it with Write, it wouldn't load it! I didn't even get a dialog box. It took me a minute even to figure out the problem. Yet the file opened fine into Edit from the ASM/EDIT system disk. What gives?  
-Gary Voth

## **It's Core Edit's Fault**

The limitation on the number of paragraphs stems from the fact that MacWrite uses a piece of software called Core Edit. This is a front end to Text Edit (in the ROM) that allows

nice things like multiple fonts and styles within words and handles justified text.

However, Core Edit is 'paragraph based'. By this I mean that Core Edit keeps a separate record for each paragraph that determines the formatting within the paragraph. Why they didn't just embed the font/style changes right in the text, I'll never know. Their way takes more space because each font/style change has to be coupled with a character position value. Oh well.

It is interesting to note that Apple stopped supporting Core Edit very soon after it was put in use. About the only thing in the world that uses Core Edit is MacWrite.  
-Chief Wizard

## **Blown 512K Macs**

Saw the note from Macowaco about how his Mac went south with a puff of smoke, a squeal and a dead analog board. Out east (DC), the Washington Apple Pi has temporarily put a hold on further group buys of the 512K upgrade kit because about 10% of the upgraded Macs have died in a similar manner. Seems as if the power supply is close to its limits. Most of the problems are power supply, some have been video. They're trying to gather statistics...Apple claims not to have any data that suggests a higher failure rate for Fat Macs. Mine's been working fine since upgrade in October, though I do notice a lot more heat out of the top left vent.  
-Paul Heller

## **Lisp Coming in April**

For all you guys interested in Lisp, ExperIntelligence will be coming out with ExperLisp for the Mac in April. ExperLisp is out at beta sites for evaluation right now. It is a fully implemented version of Lisp.

- The Terminator

## **512K Upgrade Do-it-Yourself**

If anyone is interested in upgrades ala Dr. Dobbs, the BIT9 cards are available from Ken Wilhelm here in Seattle at (206) 322-2128. A lot of people over at the U of W are converting to 512K's.

-Tim Celeski

## **LaserWriter Gossip**

It seems that product availability on the LaserWriter and AppleTalk (network) will be sometime in March. The file servers (20 and 40 megabyte versions) will be due out sometime in the summer.

-Gary Voth

## **Mac Boards**

More Mac Boards:  
NY Mac

718-643-1965

Mac Line                   401-521-2626  
MasterLink               703-476-9459  
Terry Monks              301-471-1378  
Microcom                 617-769-9358  
                              -Paul Heller

### Copy II Mac Trouble

For those of you still having trouble making copies with COPY II MAC, try this. Set both of the defaults for the internal drive. Yes it does mean some disk swapping, but for those nasty little gems that just don't want to copy in the bit copy mode, it works miracles. Probably because there isn't any problem with drive speeds.

- Jim Kimmel

### Mac Draw Loves Mac Paint

How come nobody has mentioned MacDraw .9997B? (or something like that!) You can transfer from Paint to Draw and vice-versa via the Scrapbook!

-ANT KILLER

### Window Tricks In Basic 2.0

The following applies to ALL windows that have title bars and resize buttons: If you double click on either the title bar or the size button, you will toggle the size of the window. The default sizes are the initial size (and position) that the window has at its first appearance and a full screen window. If

you manually resize a window, BASIC remembers the size when you toggle it off and on. This applies to, as implied earlier, both list windows, the command window, and the output window (actually all output windows that have a title bar and size box, even the ones you create with the program). If anyone knows any more window tricks, let me (and the rest of us) know.

-Mike Steiner

### New Version of MacPaint 1.42

Today I got MacWrite 3.95 (Jan 2, 1985) and it seems to be bug free. I also got MacPaint 1.42. I have no idea what the difference between 1.4 and 1.42 is!!

-Mac Scotty

### Mac Fortran

Even though fortran lacks many programming features that a 'C' language has, the cost of the Absoft MacFortran development system (\$300) is well worth it. Strong points are its compiler and debugger. Bad points are no modern date types and control structures.

The compiler is fast. 825 lines per minute running on my ram disk. 450 lines from a disk based system. The compiler doesn't create assembly code. Rather, four passes on the object file and walla, your application is there. An \$150 option gets you assembly source code output. Code is very compact compared to C.



## Mousehole Report

## MacTutor Vol. 1 No. 5

### OS Documentation Errors

There are errors in the OS Utilities appendix for register-based traps which return a value in A0. Bit 8 (\$0100) must be set in the trap value for such traps. Bit 8 set tells the trap handler NOT to save/restore A0. The trap address for \_GetTrapAddress is \$A146. Earlier documentation had it incorrect.

-Steve Brecher

### Hyper-Drive Shock

General Computer has mounted one of those little shock detectors that turn pink when abused on the underside of the HDA. Once the hard disk drive is installed, you can't see it. So if you drop your box, they're gonna know it and it may be a bit harder to claim warranty.

-MacGeorge

### Switching Finders Under Program Control

I have seen a few messages about alternatives to the Finder. Any application can "serve" as a finder (assuming it does the right things!!). To change the application that runs when other applications exit, you need only zap the global location FinderName.

In theory, change the contents of FiinderName to contain a string which is the name of your "finder". This could be done by a small "change finder" application which uses standard file to allow you to select the application that is to be the new finder. How about a dialog box to select between the Tardis FastFinder and the Apple Finder without the mess and permanency of changing filenames?

The file type does not have to be a "DRVR" (resource name for a driver or desk acc.); anything named FINDER (case insensitive) will do if the FinderName global contains a pascal-flavor string (counted) which is the name. The "thing" must be a runnable application, of course! The finder's creator/type has nothing to do with it. If the "system" attribute bit is set, the Apple Finder won't let you change the name of the application, and the Apple finder forces the ":system" attribute bit on for any files it finds named "IMAGEWRITER", "FINDER", "SYSTEM" and probably a few others.

Ok, I've tried it. You can change finders at will by just filling in a "Pascal" (counted) string at absolute location \$2E0 containing the filename (no device name) of the application that you want to act as the finder.

I've written an application that puts up a dialog box and lets you select either FastFinder or Mac Finder. [Hey, Bob, how about an article on this? This is great stuff!] -Bob Denny

### **Lisp Sounds Good**

ExperLisp is going to be great for the Mac even though the price is \$500. A feature of the lisp code is that it is upwardly compatible on the Symbolics 3600 series Lisp computers.

-The Terminator

### **New Finder 2.6X Features**

2.6 is the newest finder, and it has some interesting new features. When the alarm clock is on the desktop (not active), you cannot use cmd E to eject disks. It goes beep. You have to use the menu command.

RAM disk and other things that formerly showed up as being on AppleBus are now shown as on AppleTalk. Get info now shows bytes used and the number of files in use.

Under the special menu there is now a shutdown command that ejects all disk and re-boots. This is REAL handy.

Empty folder is gone, instead there is a new folder command. The put back command disappeared too. Big loss (hee hee). What did put back do? I never used it. Guess no one else did either.

There are a lot of changes to the standard file dialog. It "permits further character scrolling without additional input, automatically selects the first file, handles ejecting hard disks correctly". (How do you eject a fixed hard disk??)... "and many other minor tweaks were made", to quote Mr. Capps. An update utility SFUdate will update other disks to this new standard file dialog format.

Oh, finally, the best thing about 2.6 is that it optimizes itself for a FatMac, loading in resources normally not "pre-loaded" and only purging what it actually has to. Tremendous increase in desk accessory speed.

-Rusty Hodge, Sys Op

### **Switcher works with 2.6X**

I've had no problems with Switcher 0.90 under Finder 2.6X. I've been warned that 0.90 is the only reliable switcher. The later versions have more features, but bomb more frequently, and when switcher bombs, it usually takes the whole disk with it.

-Robert Wiggins

The switcher (1.1) problem only occurs when you try to run it from a RamDisk that has the system and finder on it. Switcher doesn't seem to check and see if the memory is assigned to something else before taking off.

-Rusty Hodge

### **Rumor Mill at the Expo**

I just got back from S.F. after an 8-hr. drive. It took me almost as long to get on the mousehole! Impressions:

- No color for the Mac in 1985, probably not in 1986.
  - No 800K floppy drive for at least 6 months.
  - No dedicated hard disk for the Mac from Apple for at least 3 months, probably 6 months.
  - "Mac XL" stands for "Mac Extra Large" or "eX-Lisa"
- The new finder should be out officially real soon!

- I enjoyed David Smith's 35-minute commercial for MacTutor, oops, I mean talk on becoming a machine language wizard. Had lots of fun at Round Table Pizza playing Maze Warz on two Mac networks with all the mouseholers at the MouseFest.

-Midnight Macker

### **Paint 1.4**

Print catalog now works the way it's supposed to. MacPaint 1.4 will now print a catalog of the last disk accessed.

-Don L.

### **Paint File In Ram**

Yes, the new Paint 1.4 will keep the whole picture in ram on a FatMac, eliminating disk access during show page and hand movements.

-The Jerk

### **FBI Playing War Games**

FBI ALERT: I heard a rumor today that the FBI is watching this board and is keeping an eye on a few people, so watch what you say.

-The Jerk

### **InLine Trap Example**

For anyone using Mac Pascal, here is a quick way to change the title of the drawing window:

**Program Sample;**

**uses**

QuickDraw2;

**const**

SetWTitle = \$A91A;

**begin**

InLineP(SetWTitle,thePort, 'title');

**end.**

This uses the undocumented "InLine" facility which gives access to any toolbox routines by its trap number.

-Don L.

### **LISA Becomes Mac XL**

Got some labels from Apple to stick over the Lisa name on the boxes, so it reads Macintosh XL. What's that tell ya?

-MacGeorge

### **Absoft Fortran**

Just got Absoft's FORTRAN in the store last week...it's Great! Makes application programs that can be run from the finder, compiles totally into 68000 code (at least that's what it says...). It even lets you save out assembled code in a text file to load in with your favorit assembler! I ran a floating point benchmark on it to see what the speed was and here are the results:

Mac..... 15.92 secs

IBM..... 170.5 secs

Applesoft.....390 secs

IBM (8087).....12.06 secs

Apple II (68000) 40.7 secs

DEC PDP 11/34 .. 27.6 secs

-The Atom

### Spinwriter 5500 to Mac

You can run a Spinwriter 5500/7700 series on the Mac serial port using the Daisy Wheel Connection Program from Assimilation Process (NOT THE NEC VERSION!). You need a different cable:

Mac: 2 > 2 :Spinwriter

3 > 3

7 > 7

20 > 19

Also connect 5, 6, 8 and 20 together on the spinwriter side. Use the NEC option in the selection box and set for 600 baud.

-Dave Smith

## Mousehole Report

## MacTutor Vol. 1 No. 6

### Sys OPs Remarks

It's anniversary time at the Mousehole! On May 12, 1985, we will be celebrating our first year of operation, and what a year it was. Back then, the assembler was only a dream [*it's still only a dream, Rusty!* ], that might be available in a very rough draft soon, and MS Basic 1.0 was our only high level language (although it was eventually followed by Forth). And old timers remember the thrill of getting a MacsBug register dump on their printer! Those were the days... (not really!)

Lately, it seems the 'Hole has been more active than usual with just an incredible amount of messages floating around. There was the high level language wars, which Consulair C won, with Lisa Pascal a close second. Then there were the hard disk wars, which are still going on. Then all the furor over the LaserWriter and its support software (like MacWrite 4.0, which still has bugs) [*Write 4.2 is the latest. - Ed* ] Then finally the thought, "Why do Mac owners think they are superior to all other computer owners?" This last one was solved easily, it is the best personal computer ever, period!

Anyways, here is a cross section of everything else of interest that happened on the Mousehole:

-Rusty Hodge  
Sys Op

### Experlisp Hardware Protected?

Now the bad news...it's really bad news! Experlisp will be copy protected via hardware! I tried to find if they were going to use the laser-hole-in-the-disk method, but they wouldn't tell me. Too bad. I wanted to put it on my Tecmar as well as backup disks.

-Macowaco

Expertelligence is using (will use) a hardware protection method that connects via the keyboard cable. Apparently they will have an intelligent BOX that will send a signal to the program when requested saying, yeh, they bought me. This way, you can move the program wherever you want (hard disk, ram disk, network) without any problem. Isn't that swell?! Well, guess we'll have to wait for someone to no-op that section of code...hee hee.

-Rusty Hodge

### Modula 2 Fluff

Speaking of MacTutor, Dave, is the Modula 2 column going to be regular? It seemed to just be (dare I say it?) just fluff this month. I hope to see more and that it will be up to the quality of the rest of your mag.

-Burrill Smith

[ *Maybe your post will encourage Mr. Bogan to dig deeper. -ed.* ]

### Draw Trouble with Laserwriter

Beware of using MacDraw with the laserwriter and pasted Paint documents. I have a 12K paint file pasted into Draw, and when printing out to the laser printer, I get a system crash after about 10 minutes. Problem is that the stack and heap meet. (At the pass?) Remember, postscript uses the stack quite heavily. However, the code must be different between Paint and Draw, since I have no trouble with printing Paint files out of the laser printer.

-The Terminator

### Inside Mac to the Supermarkets?

At MacExpo, one of the Apple Reps said Inside Mac would be released in a "telephone book" version on thin paper for about \$25, and the existing 8" manual should be shrunk down to under 1"! They joked that it would be cheap enough to sell with the National Enquirer and other tabloids at your local supermarket. ( A final printed version was due in June, but apparently is going to be late; an Apple product late? Who ever heard of such a thing?)

- Midnight

### Mac Camp at Lake Arrowhead

MacTutor is holding a Mac Camp at the UCLA conference grounds at Lake Arrowhead over the Labor Day Weekend. Three days and nights of Mac fun and classes covering C, Assembly, Basic and Pascal. Toolbox stuff too, on quickdraw, resources, clipboarding and printing. Registration is \$495 for room and board plus conference. Only 120 beds (double occupancy in cabins under the Pine Trees overlooking the lake) are available so it's strictly first come first serve; ie who gets their deposits in first! Deposit is \$250, balance due July 1st.

- David Smith

### Who's Got the Latest Version?

Switcher version 1.9 will run up to 8 applications on a 512K Mac. Andy Herzfeld is getting a 2-meg Hyperdrived Mac to experiment with. I want one too! Here's the latest versions I've found:

Switcher: 1.9  
Paint: 1.46  
Write: 4.2  
Terminal: 2.0  
Draw: 1.7  
MS Basic: 2.1 (300% faster than 2.0?)  
Red Ryder: 5.0  
MDS: A (release version?)  
Finder: 3.4

-Katz

### Comdex A Bore

MacCharlie, a PC-compatibility box for the Mac was there, two models perched behind glass. Careful inspection showed these to only be mock-ups, with "vents" on the top painted on and disk drive faceplates only. Keyboard was wood, and the keys didn't look like they work. It will ship May 25. Well, they got 60 days to make a real case for it. Price for 640K, 2-drive mode was \$1700. Apple, IBM and Commodore (as well as Atari) WEREN'T there. AT&T was, showing (Safari, 7300?) their new machine only if you signed a non-disclosure agreement! I, of course, didn't.

-Rusty Hodge

### Crash Draw 1.1

Try this to crash Draw 1.1: Select the pen (arrow) icon, then go to the pen pattern menu and drag the mouse over the top three patterns starting from left to right; while holding down the mouse button, drag the mouse all the way up into the upper right hand corner and poof! Instant freeze up!

-The Terminator

### Mac Logic Board Explained

The January issue of IEEE Computers has a very comprehensive technical/ design article on the logic board of the Mac that I am sure most people would enjoy reading. Now that I have access to a laser printer, I am really looking for a great newspaper type program so I can put pictures and words right next to each other for a Mac clubs newsletter. [Try *Page Maker* ver. 1.8 from Aldus; theirs is the best, but still buggy and in beta release. Just starting beta release, actually. -Ed.]

-The Terminator

### Back up Tank 512

To back-up ThinkTank 512, use FEDIT to change a single byte at sector 46 (relative to the start of the file), byte position 330. You should find a \$67. If so, change it to a \$60. That's it.

-Katz

### Fortran Benchmarks

Here's a benchmark for Absoft Fortran. I don't think I've posted it before. Also ran it on some other Fortran systems and other computers. It may not be the best benchmark, but it tests the features I was interested in. -The Atom

#### FORTTRAN BENCHMARKS

(UNLESS NOTED; 100 iterations)

|                    |              |
|--------------------|--------------|
| Mac Fortran        | 15.92 secs   |
| applesoft compiled | 390          |
| applesoft          | 1400         |
| IBM                | 120          |
| IBM w/8087         | 11.37        |
| DEC PDP 11/34      | 27.6         |
| APPLE W/ SAYBROOK  | 40.7         |
| MAC MS BASIC       | 720          |
| UNIVAC 1103        | 2            |
| VAX 11/782         | 1.7 (DON L.) |

Here's the program...

```
DIMENSION X(20,20),Y(20,20),Z(20,20)
 N=20
TYPE *, 'ENTER NUMBER OF TIMES TO LOOP'
 ACCEPT 100,MAX
100 FORMAT(I7)
 TYPE *, 'START TIMING'
 DO 1 I=1,N
 DO 1 J=1,N
 Y(I,J)=I+J+1.0
1 X(I,J)=I+J
 DO 3 K=1,MAX
 DO 2 I=1,N
 DO 2 J=1,N
 Z(I,J)=X(I,J)*Y(I,J)
2 Z(I,J)=X(I,J)/Y(J,I)
 CONTINUE
 TYPE *, 'STOP TIMING'
 STOP
 END
```

### Assembly Lab Improvements

Just noticed while reading Assembly Lab in #5 (April), that you consistently use:

```
MOVE <SOMETHING>, D0
CMP #0, D0
Bcc <SOMEWHERE>
```

The CMP is superflous, because the MOVE will set the condition codes appropriately. In fact, CMP #0, <anything> is always either completely unnecessary (as in the above case) or - where the condition codes haven't been set to reflect <anything> -- can be replaced with a TST <anything> instead.

Also, use the addressing modes that Motorola provides. Instead of:

```
MOVE (A0), (A1) ;TOP
MOVE 2(A0), 2(A1) ;LEFT
```



MOVE 4(A0), 4(A1) ;BOTTOM  
etc.

use instead:

MOVE (A0)+, (A1)+  
MOVE (A0)+, (A1)+  
MOVE (A0)+, (A1)+  
etc.

or even:

MOVE.L (A0)+, (A1)+ ;top, left  
MOVE.L (A0), (A1) ;bottom, right

And, instead of:

LEA mouse(A5), A0  
LEA oldmouse(A5), A1  
MOVE.L (A0), (A1)

just...

MOVE.L mouse(A5), oldmouse(A5)

-Steve Brecher

### Commodore Feeling The Waters

Although they were from a marketing company, and didn't specifically say so, Commodore was calling retail stores to see how to represent their new 32 bitter, which they referred to as "Omega". What they wanted to know was whether they should associate the name 'Commodore' with it or call it 'CBM Omega', whether the store would be interested in carrying it, were we happy with the way Apple is doing things, and comparisons between Apples, Commodores, Kaypros etc.

I told them to stay with K-Mart & Toys-R-Us.

-MacGeorge

### Bus'd out drivers

Bus'd out requires two drivers, both of which are included in the Bus'd out application file, .DDP, and .ABUS

Simply copy the two driver resources from the Bus'd Out application file and paste 'em into the System File. That's it.

Oh...for some reason the Resource Editor won't work ... use RMOVER.

- Bob Denny

### MacTraps In Filevision Format

A filevision template has come out that is a collection of the guts of Inside Macintosh. Price \$53.95. Contact MacTraps, c/o Network Nexus, Box 64, 1081 Alameda, Belmont, CA. 94002. Or call 414-591-2101.

-The Terminator



## Mousehole Report

## MacTutor Vol. 1 No. 7

### Mac Pascal Book Out

I was just down at South Coast Plaza last night and picked up a tutorial /reference book on Mac Pascal published by Computer Science Press, written by Lowell A. Carmony and Robert L. Holliday. Get this, it was all set in geneva 12 point and reduced for the book!

- The Dumacker

### Future Hughes Radar from XperLisp?

A friend and I are starting a small IR & D effort with Xperlisp on our own. The Mac stuff in Xper makes it very nice for us. The program will be Mac-like interactive and will utilize nice graphics. That will definately be handy since the project will simulate a situation radar display.

-MacoWaco

### HyperDrive I/O

4.1 is the released Finder. I was disappointed to note that HyperDrive I/O is ALL programmed; no DMA nor even interrupt-driven seeks. No point in doing asynch I/O requests to it. The Drawers Desk Accessory was written in Consulair C. For info about the author: after Drawers is in the menu bar, re-select it again from the Apple menu twice in rapid succession.

-Steve Brecher

### Tecmar and the Network?

I've got the 2.0 software from Tecmar. I'm working on getting beta software that lets the Tecmar hard disk ride the bus. Right now, it seems it isn't compatible with a LaserWriter.

- Ric

### Davong Belly-up?

There was a small article in one of the trade rags this week (I think it was CSN) that Davong (Mac Hard Disk) has filed chapter-11; gone belly up.

-THE TOOLSMITH

### MacLine IS Belly-up!

Remember the four-color fluff that Apple sent out to all the software developers, called MacLine? Well, MacLine is no more. Our subscription money was returned today with a note that they were out of business. So much for Apple's coattails!

-Dave Smith

### Local Users Groups

AMUC meets every third Wed. at 7 pm in the Fullerton Library. The Wabash Group meets every first and third Tues. at 7 pm at Wabash Computers on Kattela in Orange.

-KERRY

### **Best Mac Modem**

Muchas Gracias to everyone that recommended the Prometheus Promodem 1200. Got it yesterday, and am I one satisfied customer...it's got my vote as THE modem for the Mac.

-Davy Jones

### **Microsoft Basic 2.1**

Last I checked a week or so ago, the customer service people didn't know anything about it (or were not telling). The first one that finds out that 2.1 is released, please post!

-Dave Kelly

### **Switcher 2.0 Bugs**

I've found two problems so far. File doesn't want to quit from Switcher. File, of course, manages documents differently from most applications; it doesn't have a Save option. When quitting, it writes to the disk for some time, then hangs...

Also, if you are saving the 22K screen for each application, 2.0 doesn't error trap an install for the fourth application. You will only have about 70K of RAM after saving 3 screen maps. When loading the last application, you get a very nice special effects...

-Gary Voth

### **Switcher 2.0 and Modula-2 Bombs**

When Modula-2 is done compiling, it bombs [under Switcher 2.0]. Shift-cntrl-opt-period (Isn't that how you reset an IBM???) sometimes recovers it but not always. It doesn't matter if M2 is the only application in switcher, or even if I give it more than 128K. Since I know everyone out there is feverishly programming in Modula-2, does anyone know a way around this problem?

-Burrill Smith

### **Finder 4.0\* Works Great!**

I put six games on a disk with Finder 4.0\*, set start-up to one of the games, installed all six on the mini-finder, and trashcanned the finder. It works perfectly. It boots to the startup game, exits all games to the minifinder. Only thing is, if I set startup to Finder (then delete the Finder), it will not boot the mini-finder. It seems that for startup to be the finder, even if the mini-finder is active, the finder must be on the disk. Oh, one of the games is copy protected, and that presented no problems.

-Mike Steiner

### **Switcher and Clipboards**

When I have two applications loaded, Multiplan and Chart, I cannot carry the clipboard from one program to another. I am using the 'always convert clipboard' option from the options menu, but it still does not work.

-Jeff Goza

I tried Multiplan and Chart with Switcher 2.0 too. I get the same problem, the clipboard doesn't convert.

-Macowaco

I have Multiplan and Chart running under Switcher and have no problem transferring the clipboard. I set the 'always convert clipboard' option OFF and when I want to transfer the clipboard, I press the option key when switching applications.

-Mike Steiner

### **Finder 4.0\* Installer Problems**

Every start-up disk I insert [to be upgraded to Finder 4.0\* stuff] is judged with a "Cannot Upgrade the current Startup Disk" error message??

-Dennis Runkle

To get around the problem of disks that are rejected: 1. Boot with a different disk. 2. Change the Finder on the "BAD" disk to version 1.1g. This has worked fine in all cases. It seems to me most the disks I've had rejected had Finder 3.3X on them.

By the way, after installing Finder 4.0\*, check 50% reduction under page setup and then run the print catalog. This makes for a very nice label.

-Louis M.

### **Stop MacsBug Disk Spin**

To stop the disk from spinning under MacsBug, type: DM DFFIFF

-Bob Denny

### **Upgrade Power Supply Problems?**

I got my Mac upgraded just barely after the one bad batch of upgrades (1 in 3 failures) came out. Boy was I paranoid. But, in the interim, I have had my Mac on for periods up to 48 hours or so. Not a hitch, not a quiver, not a flake. I'd say it depends on the RAM (is it up to spec on power consumption), not the power supply. Also, the bad boards caused power supply failures in the first 24-48 hours of operation, so if you've run your 512K Mac without a problem for more than that, I wouldn't worry...

-Fred Condo

### **For a Mac Bottom, Call Mac Butt**

Yes folks, it's true, the people at Mac Bottom have an 800 number that spells Mac-Butt. I just called them and they were very nice and quite helpful. Supposed to be faster than the Hyperdrive and it lists for only \$1595. Try 515-472-9613. [There is a rumor that Apple bought them out...]

-Tim Celeski

### **CRT Gilches & Hyper's Loose Nuts**

Remember my moaning about my Mac's CRT display? Well, it finally got to where the screen was glitching about once a minute so I opened it up and found a COLD!!! solder joint on the horiz. seep ramp generator cap!! It's fixed now.

That reminds me, those of you who got HyperDrives in Jan. or Feb., open 'em up now and make sure your shock mount screws are tight. Mine backed out completely from vibration after 2 months. The drive was flopping around in the can. I called GCC and they said "Yeah, you probably got one of the last drives that wasn't glued in!" Put Loc-Tite on the screws to be sure.

-Bob Denny

### **Munched Lisp Disk**

Well guys, I have finally gotten Experlisp and in only two short hours, I managed to munch one of the two master diskettes that are sent. When I copied all of the visible files over to the Mac XL, upon booting the program, it would ask for the master disk for verification. This must be done every time the program is used. So I cleverly tried to copy over to the hard disk, an invisible file, and it just wouldn't copy, thereby munching the disk inoperative now. So I called these guys, and just like Macowaco said, they have more copy protection stuff in mind for version 2.0. Aside from the crazy copy protection scheme, it is pretty decent package and will be a lot of fun to use. (I can say that since I paid over 500 big ones for it!)

-The Terminator

### **Megamax C Praised**

Well, my vote for best C compiler is Megamax C!! The Megamax C also is much smarter in that it knows what header files to use if you forget to include them. It also happens to be a couple of hundred bucks cheaper than Manx C. I love it!

-Mac Scotty

### **Copy II Mac Ver. 3.0**

Copy II Mac version 3.01 is out...it's got a status matrix that shows which tracks didn't copy in sector mode. And it copies Mac Pascal just fine.

### **Death of the Mac XL?**

Well, I guess Steve Job's wish will come true. The Mac XL is going to be wiped off the face of the earth. My dealer told me that there is only a two-month supply of XL's and after they are gone, there won't be any more.

### **New Stuff**

Yes, version 4.1 is the official Finder. Other releases due from Apple are:

|          |      |
|----------|------|
| MacWrite | 4.5  |
| MacPaint | 1.45 |
| MacDraw  | 1.7  |

-The Terminator

### **Apple Sitting on Its Mac-Butt?**

I heard today that on Monday Apple will officially drop the Mac XL [they did !] and at the same time announce their 10 meg hard drive that sits under the Mac. Another story is that Apple bought out a company called Mac Bottom [see previous post ] that displayed the hard disk at Mac Expo '85. I guess we will see, won't we?

-Kerry

### **MDS System Released?**

You can now walk into your friendly neighborhood Apple dealer and for the paltry sum of \$195, order product number M0534, The Apple MDS development system, with Phone Book IM docs to boot.

-Don. L.

### **Lisp Bugs**

XperLisp 1.0 has some bugs. It crashes when I am using an iterate function with a bunny. Also, using CARs and CDRs to take a list apart returns lists in lower case: (CDDR '(A B C D F)) returns ;(C D f), which is a bug. For \$500, one would expect a bug-free product!

-MacoWaco

### **Mac XL Death Leaves Buyer out In Cold**

I heard about the death of the XL last Friday and hated it. That very morning before I heard the "news" I delivered an XL to a guy at Hughes who thought he was buying a machine that Apple would stand behind. I'm really pissed! Months ago I thought they were going to kill it so I always pushed 512 Mac, or a 128K if a person was financially short. A "nameless" Rep pushed us on the fact that Apple would not kill the XL. Trust away and now I'll most likely have a customer that would like his money back. I hope that an exchange for a 512K and a second drive will appeal to him.  
Probable cause for death:

Too expensive to manufacture

Low consumer demand.

Not truly compatible.

Good-looking, but too-large foot print

Not portable at 55lbs.

Not really supported by Apple, Where is the screen fix @\*%! it? The real XL

Works?

-Rob

### **The Mac That Wasn't (a Mac)**

The Mac XL was a lie from the beginning; a marketing ploy to dump unloved Lisa's on the public by doing a name plate change to fool people into thinking they were getting a Mac, when in reality it was the unwanted Lisa. I'm sure that kind of IBM-like tactic didn't sit well with Jobs. It was a very uncharacteristic snow job on Apple's part, that I for one, resented because it implied something that wasn't true. They tried to do in marketing what they had failed to do in engineering. People are very sensitive these days to being taken advantage of by computer companies that leave them holding the bag: look at Bloom County! A sorry end to the whole sordid Lisa affair.

-David Smith

*Mousehole Continued Next Page!*

## Printing and the Tecmar Macowaco

Nothing will get my Tecmar to let my Imagewriter print. Then on top of that, I tried Finder 4.1 on it and I can't copy a file onto it!

### Tecmar and Finder 4.1 Rusty Hodge

I have heard from some people that the "new" imagewriter driver and Finder 4.1 cause the Tecmar's print spooling to go out to lunch. It may be that reformatting is the only solution. Finders 3.3x and above need to be patched to have the "verify" routine no-opped.

### Finder Patch for Tecmar Macowaco

The patch for the Tecmar Finder is the following: change A002 57CA FFE2 to 4E71 4E71 4E71 using FEDIT. The second A002 found using the Hex Search is it. Eliminates the "Verify" routine used in 4.1, but not used on earlier Finder's before 3.3x.

### Rom Bug for I/O Traps (name withheld to protect the guilty)

The effect of this bug is that if you perform an I/O trap call within an SCC interrupt routine or within a VBL task, your Macintosh may bomb. Assume A0 contains the pointer to the I/O parameter block:

```
403924: LEA $360,A1 ; pointer to file system queue
header
403928: JSR *-$2E08 ; Enqueue pb (A0) on file
system queue (A1)
0392C: MOVE SR, -(SP) ; save status
40392E: ORI #$300,SR ; disable SCC and VBL
interrupts
403932: BSET #0,$360 ; set file system busy bit
```

Suppose an SCC or VBL interrupt occurs during execution of the instruction at 40392C. At this time, the parameter block had been enqueued on the file system queue, but the busy bit has not been set. If the interrupt handler performs an I/O trap, then that I/O as well as the one in the queue will be completed before the interrupt handler returns. When it does return, the file system queue is empty, and the instruction at 403932 sets the busy bit. Code following 403932 then attempts to dequeue a parameter block and execute its command code, causing a bomb.

A similar sequence of code for device drivers begins at 40118E. This code exhibits the same problem except that the timing hole is two instructions wide. Thus, if the file system or a device driver queue is not empty and the busy bit is not set, you MUST NOT perform the I/O trap.

What is the solution? Simple. If the queue is not empty and the busy bit is not set, set the busy bit yourself, perform your async I/O trap, and reset the busy bit. Hence, your I/O is

simply enqueued, and when you return, the interrupted ROM code precesses the queued I/O calls correctly. This bug is known at Apple and will be fixed in the next ROM.

### Finder 4.1 Bug Dave Kelly

While trying to do a disk copy using the Finder, the destination disk's ICON disappeared and the startup disk commenced to copy itself onto ITSELF! I restarted everything by hitting reset but the Finder was missing from the system folder. Something is kind of fishy here. Anyone else see any weird things like this?

[Yes, I have had icons disappear or move erratically under Finder 4.1. I assume it is a warning the Finder or desktop has been trashed and it's best to shut down at once. Version 4.1 definitely has a bug related to tracking icons on the desktop. -Ed.]

### More Finder 4.1 Icon Bugs MacIn Stosh

Finder 4.1 seemed to be working great until I got a "Disk needs minor repairs..." alert. When I checked the desktop, I noticed half the files had lost their icon pictures. Here's an interesting way to make your write 4.5 bomb: go to the search and replace window and hit "Command-Enter"...

### Paste Into the Calculator Ant Killer

Did you know that you can paste what's in the clipboard to the calculator and it will work just like you had typed it in? (Well, it does.)

### New Software Releases Video Whiz

Just got the new "Official" update stuff from Apple; Write 4.5, Paint 1.5, Finder 4.1 and a new combo Font/Desk accessory Mover. The new paint allows "smoothing" with the Laser to increase resolution.

### Thunderscan / Appletalk bug Brett

Thunderscan uses the battery-backed up RAM and in fact looks at a location used by Appletalk. Removing the battery for 30 seconds after turning off the computer will clear memory and allow the thunderscan to work (until you use Appletalk again).

Product description on the 20 meg disk using the external drive connector is expected shortly. [What about recent announcements that Apple was shutting down the disk project in favor of buying the technology on the outside? -Ed.]

### Building Drivers Bob Denny

To build a driver, start your assembly with a

#### RESOURCE directive:

```
RESOURCE 'DRVr' id 'XXX' atr
```

where "id" is the resource ID, "atr" are the resource attributes (usually at least system heap). The above example will give a driver names ".XXX". Following the RESOURCE directive, code the driver flags words, then the driver dispatch table. Note that the dispatch table does not need to be coded using offset. Just do this:

```
DC.W Open
DC.W Prime
... etc.
```

The linker will do the rest. (Note the linker will not resolve references across multiple .REL files in the /RESOURCES section. Your driver must end up as a single .REL file.) The key to understanding how the MDS assembler and linker handle code resources is that the linker will resolve all relocatable references against a base "address" of zero. Therefore the use of the driver routine labels as shown above will automatically translate to offsets to the respective entry points. Neat, eh? Just keep the rest of the driver "position independent".

Next link the driver with a control file like this:

```
/OUTPUT XXX.DRVr
/TYPE 'ZSYS' 'MACS'
/RESOURCES
XXX.REL
```

\$

The /type directive makes the driver image file have the "little Mac" icon like the finder and system. It's optional.

#### **Greyed-Out Text Bob Denny**

Someone asked how to gray out text. Text is grayed out by painting its enclosing rectangle with "GRAY" pattern and the pen in BIC mode. Apparently you can't just write gray text. Here is the code from the defProc for buttons, with my comments:

```
MOVE.L A4, -(SP) ; (SP) -> button rect
MOVE.L #$00010003, -(SP) ; shrink amt.
_InsetRect ; shrink rect.
```

```
MOVE.L A4, -(SP) ; (sp)=shrunken rect
MOVE.L (A5), A0 ; A0=>QD globals
PEA Gray(A0) ; (sp)=gray pat
_PenPat ; make pen gray
MOVE #patBIC, -(SP) ; bit-clear BIC mode
_PenMode ; make bit-bashing pen

_PaintRect ; BIC-out text rectangle
```

```
_PenNormal
MOVE.L A4, -(SP) ; (sp)-> shrunk rect
MOVE.L #$FFFFFFFD, -(SP) ; expand
_InsetRect ; expand button rect.
```

#### **Multiplan on the Hyper-Drive (Name withheld to protect the innocent)**

I used Fedit to move the Neil Konzen, Kensh Rutha etc. "secret" files to my HyperDrive & reset the invisible & protected bits. You can run the applications in a drawer other than the startup, they will be properly found if you click a "document" that belongs to them on still another drawer. Gad ... that's unclear ... try this:

1. Copy Multiplan to any HyperDrawer (I have an "Office Applications")
2. Make a Copy II copy of multiplan master
3. Use Fedit to turn off the protected and invisible bits on the file "Neil Konzen" on the master.
4. Copy the now visible Neil Konzen to the same HyperDrawer where you put MultiPlan.
5. Use Fedit to turn on the protected and invisible bits for Neil Konzen on the Hyper (and on your copied master, if desired).

VOILA! Multiplan will run on your HyperDrive. The same sort of thing works for other Microsoft things, Electric Checkbook & others. I went nuts till I got it sorted out, now I LOVE it.



## *Mousehole Report*

## *MacTutor Vol. 1 No. 9*

#### **Rusty Hodge Sys Op**

Lemme see, where do I start? USA Today had a poll, which brand of computer are you considering purchasing? 34% Apple, 23% IBM, rest divided. You could vote for more than one. Interesting.

MacExpo - there WILL be another MouseFest. It will probably be a get together with the people from MacBoston. There should be a survey up about this soon or post your opinions on G/eneral.

MouseHole buttons - we will be giving them out at MacExpo or you can send a SASE to Box 2323, Orange, CA

92669. One is free, if you want a lot, send some \$\$\$\$. We'll announce when they are ready. Thanks to Don for the design. Bye.

#### **New, Fledgling Macintosh BBS Mark Chally**

There is now a Macintosh oriented BBS in the Covina calling area. It is called "The Apple Bus" and it is the bulletin board of the Solely Macintosh User Group of Cal Poly Pomona...for more info about the group, and to get in touch with others with Macintosh interests, call: The Apple Bus at 818- 919-5459. Oh, there isn't any hidden password, it runs on

SnAPP, and the new user password is NEW...with your help, we can keep it public.

### **Mac Hard Disk Tom Sherman**

**Problem:** I just got back a big expense check and the bills aren't yet due. Which Mac Hard Disk do I buy?

[A] HyperDrive w/the new, Apple-Approved Installation?

[B] A 21 Meg corvus so that I can share w/my II and future Macs.

[C] Bernoulli Box

[D] The Keeper (has anyone seen one?)

[E] Wait till 1988 for Apple to deliver one.

I'd appreciate the input. It seems that I've been waiting forever! -Tom

[Consider this: If you use the network, the printer port is out since you need it for Appletalk. If you call the 'Hole, the modem port is out since you don't want to unplug the modem all the time. That leaves only the disk port. So who makes a hard disk for the disk port? Apple is rumored to be putting together a "Mac Bottm" like drive that does use the disk port (Mac Bottom doesn't!) that is not the file server product. (ie, you might see it before you die of old age.) -Ed.]

### **Journaling Discovery Don L.**

Try this: Get out any "Guided Tour" disk. With ResEdit copy the .journal driver from the system file. Paste the .journal driver into the system file of a disk you want to use journaling on. Change the driver name to journal (remove the preceding period). Quit resEdit. When your done you'll have a new desk accessory which when opened adds a menu to the menu bar with the selections Record, Play, Stop and Quit. Have Fun.

**Note:** It is important to use ResEdit to copy the driver because DA/Font mover could change the ID number and prevent journal from working. Just picked this tidbit up from the Mac Developers board on Compuserve.

### **I'm Famous! Macowaco**

I've published many times before, but this is the first time that something I had anything to do with was on the stands in B.Dalton!!! I hope they sell. [They are! -Ed.] I went out of my way and put the whole pile of Mactutors out in front and on top of the stupid fluff which was all over. Mactutor and Dr. Dobbs were the only serious programming journals there. Programming literature is big business. If you don't believe me go to your local university book store. Dave Kelly is right the laser version is outta sight. However I'm willing to admit a certain bias.

### **New Apple Product Midnight Macker**

A few days ago, Apple laid off 1200 workers. In Cupertino, they are now calling the Apple LaserWriter the Apple ResumeWriter. Oh, well.

### **Apple's New Direction? Brett**

The Chicago Area Apple Developers Meeting was held the 25th at the midwest Apple offices and here are the results... Apple announced their new direction. They said that they were turning their commercial accounts over to local dealers and getting back into the retail (not office) environment. They announced an Appletalk interface card for the Apple II and indicated that they will be getting back to supporting the Apple II market and reviving many Apple II projects that were shelved. This all comes after the big lay offs.

[If this means the holes in the Mac product line take a back seat to the ancient (and who cares?) Apple II line, we're all in trouble! -Ed.]

At the local office they had a big party because their office was not closed. Also discussed at the meeting was the Atari and Amega computers. After the presentations at the CES, and recent industry slumps, most of the developers felt that they were not as serious as we were all once led to believe. Most of the developers that had preveiously announced they would be doing prodoucts for them have now indicated that they will be staying with old Mac!

Softworks had a beta copy of their new documentation and version 2.0 of their C compiler. It is a VAST improvement over their original documentation and now indicates when a function is a K&R standard C function or a Whitesmith aberration.

We will be having a demonstration of the TMON debbuger at our next meeting. It was written by a 16 year old kid!!! He is at West Point now competing in a super math event. Our next meeting will be July 24.

### **Apple Rumblings Bob Denny**

I can't say too much about this, but believe me, there are some things happening inside Apple that are WONDERFUL. I have been talking to some of the folks who survived the purge. There has been a lot of hand-wringing in the engineering area. Many of the rookies who are responsible for some of the Mac's weaknesses have been axed. The folks who are there now are exhibiting a refreshing degree of enlightenment. One thing I will offer as an opinion ... don't expect too much in the way of major additions to the Mac product line for a while. They realize that much of what's out there right now needs cleaning up; it needs a professional polish. And they are committed to working through it and getting it clean (within the limits of the hardware limitations). More later.

### **More Apple Gossip Golden Ears**

I had lunch with and ex-Apple tech/sales-noid this week, he had a few interesting comments about the situation up in Cupertino. He feels that they are still extremely top-heavy, axed a lot of sales/ support folks but not much in the way of useless management; he also feels (hopes) that this will change very soon, as he anticipates another heavy lay-off.

He has seen the MacBottom style HardDrive, but it is evidently going to be farmed out for production elsewhere (it works on the 2nd drive port at speeds that are undocumented in IM); he has also seen a modular mac-kludge that had co-processors, plenty of memory, color and slots ... expects something around the first of the year.

Who knows, with Jobs out of the way (it was HE that was the problem for Mac and some of it's bizarre designs, NOT some lame engineers.... he squashed some of the best ideas because of HIS philosophy!) and with some action in the ][ area with the Western Designs chip, Apple could really come out things looking ok ..... BUT, things will not look pretty again until before Christmas .....

Amazingly enough, a lot of folks think that Commodore and Atari will sell everything they can build (with or without much software) and Apple will pick up the rest of the pieces. Well, Apple has a chance to clean up their act and fly right ... but they better use a little common sense for once, cause you don't get many second chances, the rest of the market will see to that ... just ask IBM. 'nuff for now, <\*\*\*ge\*\*\*>

### **MacNosy OK Burrill Smith**

Has the 5/85 Supplement been sent out? There's all sorts of references to it on MAUG, but I haven't got mine yet. I received MacNosy last night. The Mac interface is non-existent, the documentation is incomprehensible, and user-friendly is a term it's never heard of... But what a GREAT program!! If only it de-compiled it into HLL source code and added comments. Maybe the next rev?

### **Printer Driver Source? Steve the Digger**

Anyone know where to get drivers besides Apple's? [For Steve the Digger et al ... What say we draw up a petition and send it to Apple as a test of their "new openness" (to quote Sculley). The petition would be for documentation of the workings of printer drivers. UP WITH USER- WRITTEN PRINTER DRIVERS!!!!!! - Bob Denny]

### **July MacInTouch Is Out Rick Le Page**

Hi! just wanted to let subscribers know that the july

issue of MacInTouch will be in the mail next week. lots of stuff this month - review of Omnis 3 , Apple stuff, documentation (Yes David, you are in there this month), bugs, tips and much more. Thanks to all for your support!!

[P.S. I've seen this industry newsletter publication and it's terrific! One of the best such "newsy" Mac reportings in the industry. Has a welcomed technical flavor to it. Contact Rick at (617-527-5808 for subscription information. -Ed.]

### **May Software Supplement? The Toolsmith**

I called Apple's mailing operation on 7/3 to see what the &\*) was up with the May update. They said that mailing was expected to begin in about 3 weeks (or at least before August); the material hadn't arrived from Apple yet. There are 13 disks in the set being mailed. Maybe things would appear on time. if Apple used MacProject... sigh.

### **Lisp Paint Simulation Macowaco**

I'm trying to get Xperlisp to mimic Macpaint-like stuff. In drawing the object I use a "frame" quickdraw command, ie. Framerect(upper left, bottom right). Problem is when I move the mouse back towards the upper left the frames are moving to fast to be erased! Any ideas out there?

### **Lisp Solution Mike Stelner**

Try using the xor mode and drawing the framerect twice. The first time draws it and the second erases it.

### **Mashed MacTutor! The Djin**

Dave Smith. Your box has been full for days and forced to go this route to obtain an answer. The last copy of Mac Tutor I received had been so badly trashed in the mail as it was unreadable. Will they be paper wrapped or some equivalent or shall I just discontinue my subscription as I have all the fire starter material I need living in Sunny California? The Djin

[Anyone getting a messed up MacTutor courtesy of the US Postal Service, simply call Laura and she will send you a new one first class. Can't afford wrapping yet. Maybe with a few more ads...-Ed.]

### **MacNosy Mystery Burrill Smith**

MacNosy has an equate: *asicGlob EQU \$2B6*  
What's there? I can find no reference to asicGlob or \$2B6 in the MDS equate files. I'm trying to patch RamStart to work with my 1 meg, 'cause George hasn't come through with the source yet.



*Mousehole Continued Next Page!*

## Program Bombs Lisp Micro Ghoul

I have been having a problem with a consistent bomb in ExperLisp that I was wondering whether anyone else was having. The following, though a terrible little routine is a perfect way to get to the bomb (which I was finally able to isolate trying to make a nice routine, that a later version of this piece of trash was working towards). Well, enough about that, here it is:

```
(SETQ MYWIND (NEWGRAFWINDOW '(100 100 300 350)))
(MYWIND 'SETWTITLE "GONNA BOMB!!")
(DEFUN COORDINATES ()
 (GETMOUSE))
(WHILE
 (NOT (BUTTON))
 (MYWIND 'SELECTWINDOW)
 (APPLY
 FILLOVAL '(50 50 (CAR (COORDINATES)) (CADR
 (COORDINATES)) MYWIND)))
```

Please excuse any typos or forgotten "(" or ")" or "", if I forgot them here does not mean that I forgot them in the actual code, more important is the fact that though Lisp tries to give me an error it bombs with the ID #2 everytime. In fact the first time rather than just a bomb it added the desire to have a disk swap (though both disks where in their respective drives!)

## Contest Winner The Catcher

The Catcher (alias Steve Yaeger) wins the Mousehole printing contest sponsored by MacTutor, with this assembly entry to mimic the BASIC statement: PRINT "HELLO":END on the Macintosh. Steve wins a check for \$50 and a MacTutor thank you. DON L also submitted an excellent answer that we will print next month.

```
; Print "hello" on the printer.
; Written by Steve Yaeger
; If the printer is off-line, this will hang the Mac
; until it is put back on line
; If there is no Printdriver file it just Beeps and exits
;
; The Linker file looks like this
;
; HELLO.REL
;]
;
; These are the Equates Notice that the iPrReset, iPrLineFeed,
; and iPrPageEnd are Longwords and not words as in
; PREQU.TXT from Counsulair.
```

XREF START

; These are the PrDvr constants.

```
iPrDvrRef EQU $FFFD ; Driver's RefNum = NOT ResID
iPrIOCtl EQU 5 ; The Raw Byte IO Proc's ctl #
PrDevCtl EQU 7 ; The PrDevCtl Proc's ctl #
iPrReset EQU $00010000 ; The CParam for res
iPrPageEnd EQU $00020000 ; The CParam for end
iPrLineFeed EQU $00030000 ; The CParam for pap
IParm1 EQU 0 ; the three printer parameters
IParm2 EQU 4
IParm3 EQU 8
```

; These are the standard call parameters

```
csCode EQU $1A ;control dependent command
csParam EQU $1C ;control dependent param.
ioCompletion EQU $C ;pointer to async notifier routine
ioResult EQU $10 ;returns operation results
ioFileName EQU $12 ;pointer to name of driver
ioRefNum EQU $18 ;driver reference #
ioPermssn EQU $1B ;read/write permission
```

; Traps used in this program

```
.TRAP _InitGraf $A86E
.TRAP _InitFonts $A8FE
.TRAP _FlushEvents $A050
.TRAP _InitWindows $A912
.TRAP _InitMenus $A930
.TRAP _InitDialogs $A97B
.TRAP _InitCursor $A850
.TRAP _TEInit $A9CC
.TRAP _Open $A000
.TRAP _Control $A004
.TRAP _SysBeep $A9C8
```

START

```
;Initialize Managers
PEA -4(A5) ;Space Created For Quickdraw's Use
_InitGraf ;Init Quickdraw
_InitFonts ;Init the Font Manager
MOVE.L #$0000FFFF,D0 ;This Mask Is For All Events
_FlushEvents ;Flush All Of These Events
_InitWindows ;Init the Window Manager
_InitMenus ;Init the Menu Manager
CLR.L -(SP) ;restart procedure
_InitDialogs ;Init the Dialog Manager
_InitCursor ;Init to arrow cursor
_TEInit ;Init Text Edit
```

;----- OPEN THE PRINT DRIVER -----

```
LEA '.Print',A1 ;Get a Pointer to the Drivers Name
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.L A1,ioFileName(A0) ;Store the Drivers Name addr
CLR.B ioPermssn(A0) ;fsCurPerm (whatever we got)
_OPEN
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.WioResult(A0),D0 ;Check for an error
CMP.W #0,D0 ;is result ok?
BEQ @0 ;Yes skip the error bell
MOVE.W#30,-(SP) ;beep length
```



```

_SYSBEEP ;beep to indicate error
BRA @4

@0
;----- RESET THE PRINTER TO DEFAULTS -----
LEA IOPARMS,A0 ;Get the Parameter base addr
MOVE.W #iPrDevCtl,csCode(A0) ;Set up a control type call
MOVE.L #iPrReset,csParam+IParam1(A0) ;only 1 (Reset)
CLR.L csParam+IParam2(A0)
CLR.L csParam+IParam3(A0)
_CONTROL
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.W ioResult(A0),D0 ;Check for an error
CMP.W #0,D0 ;is result ok?
BEQ @1 ;Yes skip the error bell
MOVE.W #30,-(SP) ;beep length
_SYSBEEP ;beep to indicate error
BRA @4

@1
;----- ACTUALLY PRINT THE WORD HELLO -----
LEA TEXT_STRING,A1 ;Get address of string to pr
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.W #iPrIOCtl,csCode(A0) ;Set up a text streaming call
MOVE.L A1,csParam+IParam1(A0) ;Parameter one is a
;pointer to the text
MOVE.L #(TEXT_STRING_END-TEXT_STRING),
csParam+IParam2(A0)
;Parameter two is the length of the text
CLR.L csParam+IParam3(A0) ;No third Parameter
_CONTROL
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.W ioResult(A0),D0 ;Check for an error
CMP.W #0,D0 ;is result ok?
BEQ @2 ;Yes skip the error bell
MOVE.W #30,-(SP) ;beep length
_SYSBEEP ;beep to indicate error
BRA @4

@2

```

```

;----- PRINT A CR+LF JUST FOR THE FUN OF IT -----
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.W #iPrDevCtl,csCode(A0) ;Set up a control type call
MOVE.L #iPrLineFeed,csParam+IParam1(A0)
;only 1? parameter (CR+LF)
CLR.L csParam+IParam2(A0) ;Parameter 2 undetermined
CLR.L csParam+IParam3(A0)
_CONTROL
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.W ioResult(A0),D0 ;Check for an error
CMP.W #0,D0 ;is result ok?
BEQ @3 ;Yes skip the error bell
MOVE.W #30,-(SP) ;beep length
_SYSBEEP ;beep to indicate error
BRA @4

@3
;---- EJECT A PAGE SO THAT WE CAN SEE WHAT HAPPENS
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.W #iPrDevCtl,csCode(A0) ;Set up a control type call
MOVE.L #iPrPageEnd,csParam+IParam1(A0)
;only 1 parameter (eject)
CLR.L csParam+IParam2(A0)
CLR.L csParam+IParam3(A0)
_CONTROL
LEA IOPARMS,A0 ;Get the Parameter base address
MOVE.W ioResult(A0),D0 ;Check for an error
CMP.W #0,D0 ;is result ok?
BEQ.S @4 ;Yes skip the error bell
MOVE.W #30,-(SP) ;beep length
_SYSBEEP ;beep to indicate error
@4
RTS
; Data Structures
TEXT_STRING DC.B ' HELLO'
TEXT_STRING_END
.ALIGN 2

IOPARMS DCB.L 20,0
;----- end of program -----

```



## Mousehole Report

## MacTutor Vol. 1 No. 11

### Pagemaker Boo-Boo MacGeorge

Dave must have them pretty shaken up at Aldus with all his bug reports [during beta testing -Ed.]. It seems Aldus has a batch of Pagemakers 1.0 shipped to dealers which contain nothing but demo disks (I got 2). If you buy a Pagemaker, check the system disk at the store and make sure it's not a copy of the demo!

### Long Beach Apple Fest Mousehole Rusty Hodge

Okay, here goes: Sunday, 15 Sep, 7:30 pm the (and I quote) "MouseHolePizzaFest" (yes, 1 word), is set for Me'n'Ed's Pizza. Bring your own power strip. More later this week.

### Apple's Hard Disk Rumors Mac Scotty

I talked to someone at Apple today and was told that the new 20 meg hard disk [from Apple] will be released sometime around Sept. 16. The price is around \$1600! The sales reps are going in for training on Monday!!

### Helix Versus OverVue Bob Denny

Use Helix! OverVue is a distant 2nd for small jobs. By the time you get good on it, version 2.0 (I am beta testing it now!) will be out with 25 new tiles, picture fields and maybe some speed-up. You won't regret it.

[Bob Denny broke his arm riding a horse last week and typed this post with great difficulty, left-handed. Get well cards

may be sent to him care of MacTutor. -Ed.]

### **Bizarre Occurrence Fred Condo**

Chalk this up to weird stuff: My internal drive suddenly decided that all disks, formatted or not were unreadable, and that, furthermore, nothing could be formatted. In a last ditch attempt to allude an Apple Dealership's inflated repair prices, I yanked the battery, thereby clearing the parameter RAM. A MIRACLE! Yup. Fixed it! Now all I wanna do is get my hands on the programmer whose program (I don't know which it was) fooled with my sacred parameter RAM...

### **More on Databases Deirdre L. Maloy**

So far as I know, Omnis 3, Helix and MacLion are currently the only RELATIONAL databases. Omnis 3 really isn't as powerful as it is billed to be; MacLion has a powerful but frustrating programming language; Helix is very Mac like. I use MacLion myself (c'mon, how many databases give you toolbox access??)

### **Japan Macs MacGeorge**

Next time you are in Japan, check out their Macs. In order to fit the Japanese Kantana into the Mac, Mac's in Japan come with Hyperdrives from Apple!

### **Searching Resource Files Burrill Smith**

I have a resource file and I want to see if it contains a CODE resource with an ID of 1. If I do an `_OpenResFile(Filename)` and then a `_GetResource( Code, 1)`, the resource manager searches all the resource maps. I can do a `_HomeResFile` and see if the CODE 1 resource it finds belongs to me, but I'd rather restrict the search to just my resource file. `_UseResFile` restricts the search, but since my resource file was opened last, it gets eliminated from the search first. Is there a simpler way?

There is no simpler way. Why the concern about restricting the search (the maps are all in memory)? Note that if you only want to know whether a certain resource is in a file, but don't want to actually load it into memory, `_SetResLoad(False)` before calling `_GetResource` and then `_SetResLoad(true)` when you want to return to normal resource operations.

-Steve Brecher

### **Microsoft's Typographer Tim Celeski**

Typographer, for those of you that are interested is a new composition program from Microsoft. I've been alpha and beta testing the program for some six weeks now. Cost is a major difference between it and Pagemaker as it is \$195. It has unlimited document size, whereas Pagemaker is limited to 16 pages. It has increased ability to deal with headers and footers, automatic baseline jump, adjustable word spacing (kerning

though slightly limited in this release), footnoting capacity, indents, tabs, search and replace and increased ability with hyphenization. I think it has a very nifty way of re-fitting text, which is a little awkward with Pagemaker. It is as simple as drawing a box; it fills with text - if you don't like it, you simply change the size or shape of the box, and it re-fills it!

The real comparison is how is it different to use? From my own graphic designer's point of view, Pagemaker is easier to design in. Typographer's text handling is quite sophisticated (it doesn't use word wrap, rather it uses a much more sophisticated technique common to real typesetting equipment - the author wrote software for Allied linotype - consequently the whole line and each word in it is taken into consideration in regards to spacing). It was a sensation at the TYPE X typesetter's convention for it's precision. People with heavy text handling will gravitate to Typographer. It has some advantages for manuals, books and other heavy documents. Another advantage may be speed - it was done entirely in assembly. I use both Pagemaker and Typographer daily and urge everyone to check them out.

### **Draw Bugs MacoWaco**

When I group two boxes with some text and then move the group around, the text shifts and changes position. I then have to ungroup and realign the text. Is this a bug or what?

You aren't dreaming. I detest that text positioning bug. It's a crippling problem. It also shows up if you duplicate a group containing text objects several times (to place multiple copies at evenly spaced locations, right?) The damn text drifts out of line. No way to use grid either, as it doesn't seem to snap to it.

-Bob Denny

### **MacFest Boston Report Steve Brecher (via MAUG by Tom Dolby)**

Here's a few things at Boston that caught my eye: Challenger showed a MacDraw like program in 3-D that allowed stretching a whole side of a house from plane or vertex of lines. Mac to midi interface with overlay of up to 32 tracks on the Mac with perfect replay. Outrageous! Typographer from Microsoft that looks like MacPublisher and Pagemaker fused together. Abatron was the hit of the show with a conversion program that translates Apple 6502 assembly programs into Mac 68000 programs automatically. Only ten minutes to create a running Mac version of an Apple game. Bill atkinson was sitting on the floor in front of me and was impressed as I. Apple games Robotron and Sabotage were converted from scratch. Abatron also showed a Scantron 300 for \$2400 that scans at 300 dots per inch (like thunderscan) and makes postscript, paint or pagemaker compatible output file for the laserwriter. Microconversion has a four megabyte RAM board that is Hyper compatible! The new 20 meg hyperdrive was shown (20% speed increase).



### **Amiga Reaction Rusty Hodge**

What do you think of the Commodore Amiga?

- (1) Commodore what???
- (2) Sorta like a toy Mac - like it but not serious
- (3) Love it!
- (4) Don't like it (generic negative reaction)

Mousehole Voting record: 55 total votes to date

1. 5%
2. 52%
3. 23%
4. 18%

### **Disk Port Versus Serial Port Steve Brecher**

The external disk port has a transfer rate of 500 Kbits/sec. The serial ports with external clocking have a nominal maximum rate of 1000 Kbits/sec and an effective rate of 700-800 Kbits/sec. So don't wait for a disk port type hard disk (ie Apple's) if speed is your criterion.

### **Table Driven Event Loops Bob Denny**

Is there any way to statically initialize an array with pointers to functions? The reason for this is that with a setup like this, it would be pretty easy to write an event loop so that it could simply index the pointer array and make the proper call.

One of my favorite things to do with C is to write table

driven software. Routine pointers are indispensable to this sort of effort. The Mac requires position independent code, which makes the job more difficult. But it CAN be done. Normally routine calls are done PC-relative (intra-segment) or via the CODE-0 jump table (inter-segment). With a table you must "fix up" the routine addresses at program startup or there is always a chance the code segment may move (relocatable); you need a way to call PC-relative. One way to do this is to put an index in the table, then use a case statement to dispatch. This would work with inter- and intra-segment calls.

Consulair Mac C will correctly handle a table of pointers to functions. It's library routine that initializes global data treats such function pointers properly. The functions referenced in the table should be in the first segment if you have more than one segment. For example:

```
void foo1() {}
void foo2() {}
void foo3() {}

void (*table[])() = { foo1, foo2, foo3 };

void dispatch(funcnbr)
 int funcnbr; /* 0..2 */
{
 (*table[funcnbr])();
}
```

The table is allocated in the application global area ("below A5") and initialized automatically at startup. Other C compilers probably do this ok too, but Mac C is the only one I've tested.

- Steve Brecher

---

## *Mousehole Report*

---

## *MacTutor Vol. 1 No. 12*

---

### **Finally Aboard! Jim Fitzsimmons**

After attending two Mousehole pizza parties and meeting numerous Mouseholers from all over the USA...I have finally figured out how to log on this infamous board thanks to a step by step from Rusty (Thanks Ruusty I have a SHATTER comic book for you). Sooo, greetings one and all. I'm delighted to be a member of this exclusive gathering of wizards and heavy users not to mention all you hackers. P.S. Still learning to type & Spell.

### **New Prices Mike Steiner**

New Prices and stuff. 1) The 128K Macintosh is dead. Apple removed it from the price list and told dealers to upgrade any 128K's left in stock. 2) The 512K Macintosh is now \$2499. That is just 4 bucks more than a 128K cost when it first came out. 3) There is a new Imagewriter driver for the Imagewriter II (It will also work with the Imagewriter I). There is a new Font/DA mover that fixes the bugs in the first one when working with a 128K Mac. 4) Through the

Looking Glass is now free. Apple is not charging dealers for it and recommends no charge to the customer. 5) Hyper 20K is now \$2195 and the Hyper 10K is \$1695 (or \$1595; I forget which).

### **Mass Storage Technology Toothy Macs**

Those of you hardware techies who stay up with new developments-- what do you think? The following is from a (not new) July 22 EE Times: "Eastman Kodak Co.'s Verbatim Division...is poised to become the first supplier of erasable optical-disk drives for OEMs. "...Verbatim demonstrated a working prototype of a 44-Mbyte optical drive that uses removable 3 1/2 inch media."

The company says OEMs will pay less than \$300 for the system. It will be offered at the end of 1986 in various capacities, ranging to well over 100 Mbytes. Media prices should be about \$35 for a removable 3 1/2 inch disk. "...featuring a 5-Mbyte-per-second speed...the laser and biasing magnetic head prerecords an entire track as digital ones. The laser then is turned on and off at a 5-Mbyte per second rate, simply erasing these data bits to form zeroes

wherever needed. Such a burst mode can record and read data at faster burst speeds than the track-sector system. "Bate is optimistic that 3rd party standard interfaces will be available Winchester market. "'We're going to be at every significant conference to give people an update on this drive,' said Bate, 'It's a living, breathing reality.'"

So here's the question: Why isn't news of this thing filtering down? I read LOTS of trade mags: and the talk is always about CD ROM and WORM. If we're not careful we're likely to have to live thru a whole generation of ROM stuff just because that's what Sony et al are geared to produce for us, when there's 100 MByte, 5 MegaByte/sec read/write/erase technology that's so cheeeep it can be sold at \$300 a crack. Input, anybody?

### **CD ROM Chief Wizard**

In my opinion, CD ROM is being VERY overrated. First of all, when you stop to think how many units a company has to sell to make a profit after they've entered ALL that data and masked a CD ROM, it's outrageous. The actual ROM reader hardware is easy; it's very similar to your brand new CD player at home. But people have not thought out the problems of the media. How many large databases do you know of that DON'T EVER CHANGE? There's talk of putting dictionaries and encyclopedias on CD ROM. Every dictionary and encyclopedia that I know of has a new edition or at least updates every year. After two or three years, can you see someone searching the main ROM, then searching 3 addendums? I can't think of ONE large, electronic database that doesn't get updated at least once a month. I really don't think people will have a use for nonalterable mass storage devices or media. As soon as a single producer comes out with a system that can do the same amount of data in an alterable format, CD ROM will die.

### **Apple 20 meg HD prices Jcom**

The cheapest price for an Apple 20 meg HD is \$929. Don't get too excited that's the price BIG buyers can get. Hughes aircraft will buy about 7.5 million dollars worth of Apple gear next year and qualifies for a schedule 3 discount (the max). This number may be useful to you in terms of driving your best deal. After the rush wears off I think any bargainer will be able to get the HD for around \$1000. At that price, color me interested.

JCOM, Maybe you should become a certified developer. My price as a developer is \$750.00. The hard disk is going to be

available in limited quantities at the end of the year (Nov or Dec). I got this info from Apple Developer Relations. Still waiting on the repair of the Hard Disk for the MouseHole Download. Hope to have it up and running soon. -Katz

### **Microsoft/Switcher Mike Steiner**

Microsoft's official recommendations for memory sizes for MS products and Switcher:

| Application | Preferred Size | Minimum Size |
|-------------|----------------|--------------|
| Excel       | 304K-512K      | 256K         |
| Chart       | 192K           | 128K         |
| Multiplan   | 160K           | 128K         |
| Word        | 160K           | 128K         |

### **Finder 4.49 Micro Ghoul**

I have been playing around with Finder 4.49 (and the software for the 20 meg RAM based boot) and have experienced very strange problems with 2 MB mode! Does anyone have even a clue as to whether or not there is some reason as to why this situation exists? By the way for those of you who have not had a chance to play with it, it is hot! LOVE the new file package and real folder system!

### **PageMaker and MacDraft and Laser Bob Denny**

The combination of PageMaker, MacDraft & Laser is a SQUIRRLY combo! After three thrilling days of trying to tame these guys, here are some nifties:

If it looks good on the screen at 4X mag, it'll look bad on the laser printed from draft. Holes, mis-aligned lines, horrid arrow heads. Group Draft pic, cut to scrap, paste in PageMaker, more wired things. The pic changes again, things don't line up, different from the Draft wierdness, laser output is sad. If you don't set "ink" on text blocks (that's right, text block), to NONE, you'll get big lines extending to the right & down from the upper left corner of each Draft-generated text block when pulled into PageMaker & laser printed. They don't show on the screen, just the printed page. Try to cut from Draft into Draw ... most things don't show up under Draw, even though the scrapbook says that the cut item consists only of a PICT resource, should be digestible by Draw. Some things that Draw cannot create show up fine when pasted in from draft. Getting arrowheads to work from Draft is impossible. And Draw as those GIGANTIC arrowheads on thick lines.



*Conclusion Mousehole, Next Page*

## ROM Serial Driver Help

Michael Carlton

I'm trying to open the modem and write to it with Megamax C. So far I have been very unsuccessful. It always hangs the first time I try to send stuff out. I'm using:

```
opendriver(".AIn", &in);
opendriver(".AOut", &out);
pbwrite(&paramblock,0);
```

after filling in all the blocks of paramblock. This always hangs. Is this the right way to do it? IM isn't very clear about ROM drivers. [ We are looking into the ROM serial drivers. Look for an article in the Jan. issue. -Ed.]

## More Draw 1.9 Complaints

Bob Denny

Create a box, type some text while the box is selected, creating a centered text in the box. Group it. Then grab the grouped "object" with the arrow and drag it around in circles, 5 or 6 times. [watch what happens to your text!] You tell me... did they really beta test this software?

## QUED Editor 1.21 Is Out

Steve Brecher

QUED ("Quality Editor for Developers") version 1.21 is now available. Software Supply at 4618 E. 6th St. in Long Beach (me) is a dealer for QUED. The new version provides windows that can be split horizontally and/or vertically into independently scrollable "panes", synchronized scrolling, show invisibles command for control characters, ascii codes command showing characters in their ascii numeric code, tabs and returns used as literals in Find/Change dialogs and it's compatible with the MDS Exec system for those using MDS. QUED is \$65 plus tax. Updates free to 1.1 owners.

## Death of Typographer?

Tim Celeski

Microsoft has dropped the program [Typographer]. For those of you that don't know what Typographer is... A text and graphics composition program somewhat on the order of Pagemaker though much different in design. It has a pretty impressive technique for handling the size and shape of text blocks and  *Kerning an unlimited document length* that made it particularly suitable for manuals, catalogs, etc. Typesetters loved it as it was precise and versatile. Alas, Typographer had problems... bugs, bugs and more bugs... so Microsoft has decided to drop it and Broca Software, the originators may come out with an enhanced version in the spring. I hope so. Though not as easy as Pagemaker to design in, it did have a number of valuable characteristics...

## AppleTalk Release 39

Bob Denny

There is a new release of the AppleTalk drivers on CompuServe MAUG (MACDEV PCS-7 DL8... how 'bout that for givverish!!). Anyhoo, there is a MacPaint file ATLK39.BIN and INSTAL.APP is the new installer. Minor changes for inter-netting (Apple actually tried it!)

## Basic Compiler?

Gary Voth

Sorry, but the rumored compiler for Microsoft Basic is not yet available to the public. The closest you can get is a 2.1 runtime interpreter, which will allow you to create a double-clickable stand alone program [runtime license is \$250 per year!]. The 2.1 upgrade is said to be somewhat faster than 2.0 however.

## MacMemory Board Is OK!

Michael Carlton

Memory upgrades by MacMemory to 1.5Mb are a great system. I'm running on one and I love it. [So do I! -Ed.] Features include: 1. Ram disk software that gives you a choice of 1 meg ramdisk with a 512K Mac or a 1 meg Mac with a 400K ramdisk. 2. No roms, rom patches or rom changes needed. This system will work with the new Apple roms, giving you a contiguous 1536K for heap space. [But will the software program they supply need updating? -Ed.] 3. The ram is recognized by the Finder and other applications. 4. No fan is required, as the board adds only 1.3 watts to power used. 5. Best of all, the ramdisk is NON-VOLATILE! You can reboot and recover everything on the ramdisk after a system crash as long as you don't turn off the power. [This feature has been particularly great for me; in fact, it may be safer than disk since I've lost files written to disk in MacWrite when Write crashes, whereas, if it gets on the ramdisk, a crash doesn't bomb the file. -Ed.]

## What Is MacScheme?

Macowaco

Macscheme is much more than Common Lisp. While it is true that Scheme and Common Lisp look the same and that Macscheme can do anything ExperLisp can with regard to LISP stuff (not graphics, windows or menus), Macscheme contains First Class Functional Objects. . These allow one to have an infinite number of functions that are always available. It also allows functions to make functions to make functions to make... etc. It is really heavy duty. Three major universities have already made it as the standard Lisp for their CS depts. The toolbox access is coming soon too!



# Author's Index

Prepared by Richard Schroedel

Author: Alviani, Frank  
    Asm: Resource Formats in Assembly  
Author: Austin, Kirk  
    Asm: Midi Software Connection  
    Asm: The Midi Connection  
Author: Bogan, John  
    Modula 2: Modula's History & Benchmarks  
    Modula 2: Towers of Hanoi, part 1  
    Modula 2: Towers of Hanoi, part 2  
Author: Bouldin, Chuck  
    Fortran: Fractals in Fortran 2.1  
Author: Brecher, Steve  
    Pascal: I/O Completion  
    Pascal: Pascal GENERIC Proc  
    Asm: HFS File Structure  
    Asm: Tech Questions  
    Asm: Screen Sleep  
    Asm: Text Edit  
Author: Coad, Timothy T.  
    C: Dial a Fortune Apple Talk Server  
Author: Cohen, Andy  
    Lisp: 3-D Rotation  
    Lisp: An object oriented language  
    Lisp: CONS, CELLS, and Quickdraw  
    Lisp: Functions in Lisp  
    Lisp: Introduction to ExpertLisp  
    Lisp: Iteration Techniques  
    Lisp: Recursion and Windows  
    Lisp: Windows and Tic-Tac-Toe  
Author: Denny, Robert B.  
    C: A New World  
    C: Application Template in C  
    C: Pict Rotation with CopyBits  
    C: Dial a Fortune Apple Talk Server  
    C: Text Edit & Scrolling Dynamics  
    C: Function Resources  
    C: Programmer's Guide to Networking  
    C: The Mac Window Technology  
    C: Custom File Type Dialog Box  
    C: Planning for Desk Accessories  
    C: Vertical Retrace Manager  
    C: Window Dynamics  
Author: Derossi, Chris  
    Pascal: In Line Traps Create Windows!  
    Pascal: Introduction to Quickdraw  
    Pascal: MacPascal Arrives!  
    Pascal: Ports O' Call in Quickdraw  
    Pascal: Quickdraw does regions!

Author: Eugenides, Jan  
    Asm: Grow Windows in MacAsm  
Author: Flott, Rick  
    C: Arrows in C  
Author: Jasik, Steve  
    Asm: Jasik Hypes MacNosy  
Author: Kelly, Dave  
    Basic: Alphabet Soup  
    Basic: Library routines for disk eject  
    Basic: Anybody know what Date it is?  
    Basic: Fatbits approach to cursor editing  
    Basic: May I have your order please?  
    Basic: Play Icon Concentration  
    Basic: Poke Screen Memory Direct!  
    Basic: Printing Techniques  
    Basic: Printing Techniques for Basic  
    Basic: Reading Paint Files  
    Basic: Rectangle Utility for Controls  
    Basic: The Art of Clipboarding  
    Basic: The Standard File Dialog from BASIC  
    Basic: What Light Through Yonder Window  
Author: Kichline, Van  
    C: C Glue Routines for Filter Procs  
Author: Langowski, Jörg  
    Forth: A Forth Decompiler  
    Forth: Controls From Forth  
    Forth: Curve Fitting, Part II  
    Forth: Curve Fitting, Part III  
    Forth: Disk Directories & File Editing  
    Forth: Forth Blocks to Ascii Text  
    Forth: Forth Goes SANE  
    Forth: Inline Code Speeds MacForth  
    Modula 2: Matrix Multiplication  
    Forth: Principals of Text Editing  
    Forth: Forth Questions & Answers  
    Forth: Recovering Lost Files  
    Forth: Solving Systems of Equations  
    Forth: Traps and so FORTH  
Author: Lieberman, Bruce  
    Asm: A Communications Primer  
Author: McBride, Mark  
    Fortran: A Shell Application for 2.1  
    Fortran: Random Generator in MacFortran  
Author: McGreggor, Keith  
    C: MacPaint File Formats in C  
Author: Mitchell, Jeff  
    Asm: Mac Pals & Serial Ports  
    Asm: Mac Memory Explained

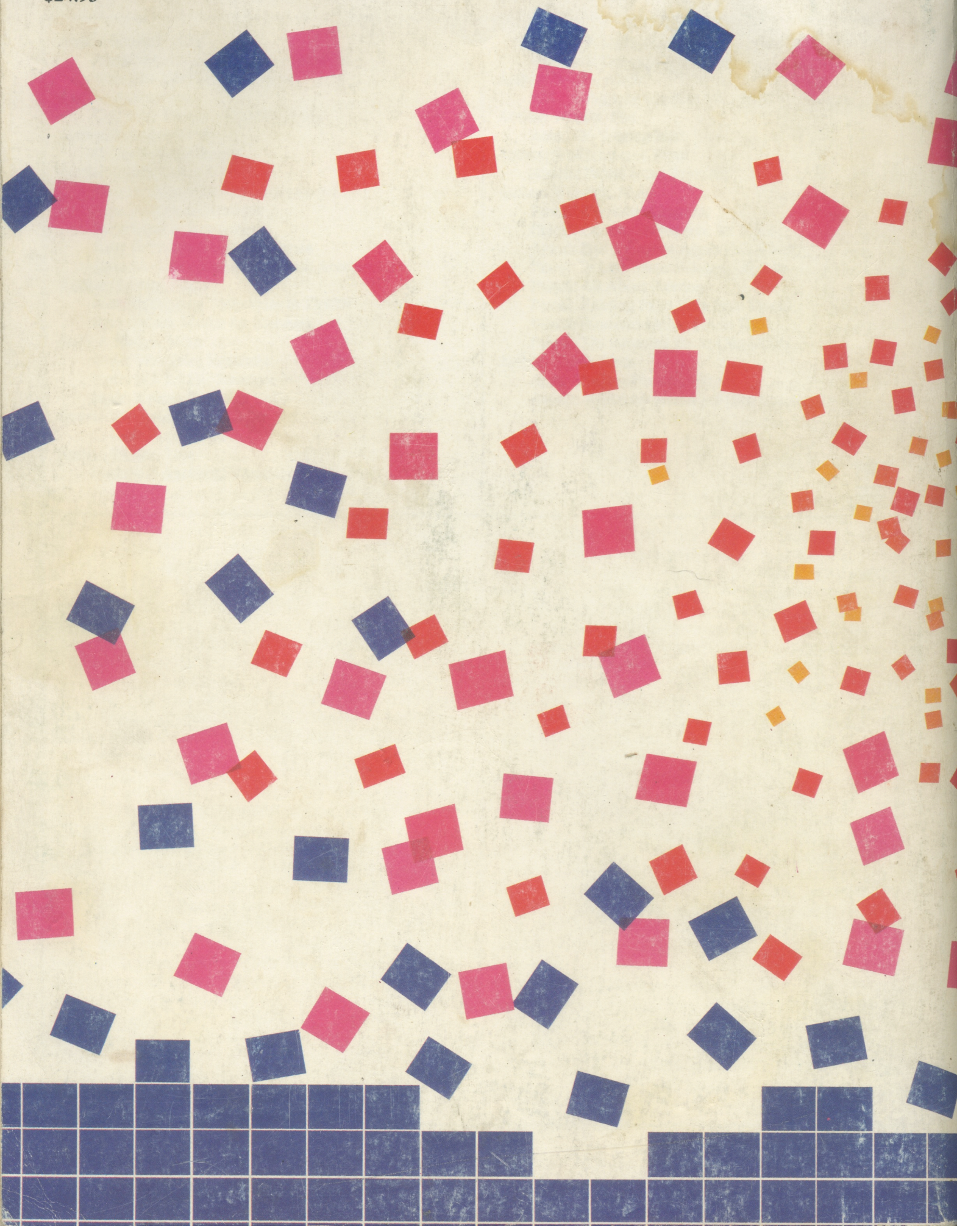
Author: Morton, Mike  
Pascal: Mondrian 1-13 (See Volume II)  
Asm: The DissBits Subroutine  
Author: Pence, John  
C: C Glue Routines for Filter Procs  
Author: Schuster, Mike  
C: Try Pop-Up Menus!  
Author: Smith, David E.  
Asm: A Double-clickable start-up Ad  
Asm: Animation Example  
Asm: Icons as Resources  
Asm: A Simple Window Program  
Asm: Modeless Dialogs and Desk Accessories  
Author: Snively, Paul F.  
Asm: Debugging Techniques in TMON  
Asm: PrLink Source for MacAsm  
Author: Steiner, Mike  
Basic: Mac Paint Simulator  
Basic: MS BASIC Structure  
Basic: Rescue Protected Basic  
Author: Swannack, Wesley  
C: Desk Accessories in C  
Author: Taylor, Tom  
Modula 2: Anchored Variables

Modula 2: Using the Alternate Screen Buffer  
Modula 2: Debugging with Macsbug  
Author: Weaks, Allyn  
APL: A Beginners look at Mac APL  
Author: Weston, Dan  
Asm: The Talking Mac  
Author: Wollschlaeger, Peter  
Article: Pascal Icons Revealed  
Author: Wootton, Alan  
Pascal: Christmas Graphics  
Pascal: Custom Dialog Box for Input  
Pascal: Dial a Fortune Apple Talk Client  
Pascal: In Line Traps simulate OS calls  
Pascal: All About Printing  
Pascal: Reading Paint Files from Pascal  
Pascal: Resource Utility DA in TML  
Pascal: The Amazing Pic to Clip Utility  
Author: Yerga, Chris  
Asm: A Micro-Finder Utility  
Article: WIred for Sound 1-9 (See Volume II)



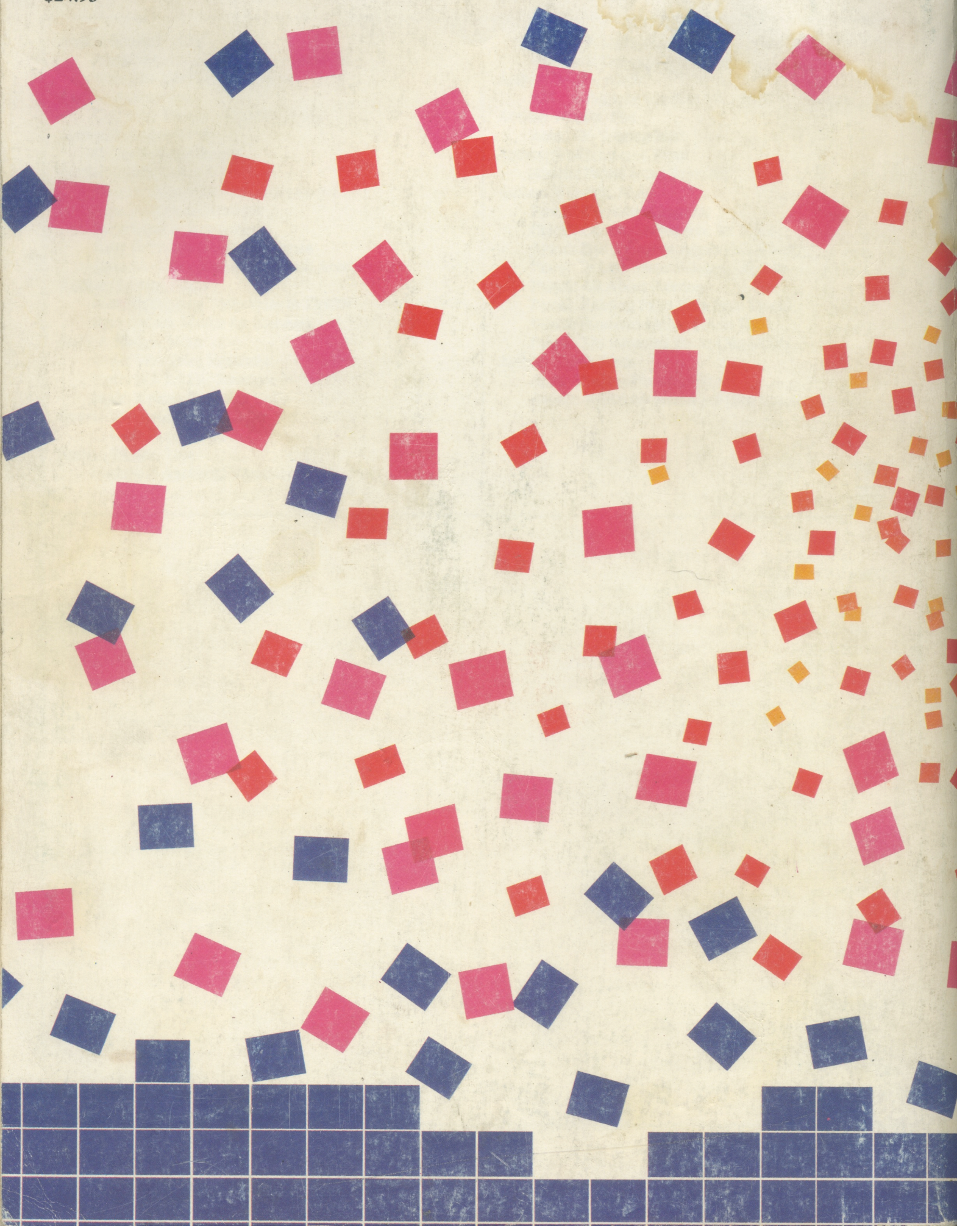


\$24.95





\$24.95





\$24.95

